Courant Institute of
Mathematical Sciences

HIGHER LEVEL PROGRAMMING

Introduction to the Use of the Set-Theoretic Programming Language SETL

R.B.K. Dewar, E. Schonberg and J.T. Schwartz

PRELIMINARY TRIAL EDITION - SUMMER

Courant Institute of
Mathematical Sciences
Computer Science Department

New York University

"HIGHER LEVEL PROGRAMMING"

Introduction to the use of the
Set-Theoretic programming language SETL.

by R. B. K. Dewar
E. Schonberg
and J. T. Schwartz

PRELIMINARY TRIAL EDITION -SUMMER 1981

Computer Science Department
Courant Institute of Mathematical Sciences
New York University

WARNING: THE PROGRAMS IN THIS TEXT ARE NEITHER DEBUGGED NOR COMPLETE. THEY WILL APPEAR IN COMPLETED AND TESTED FORM IN THE NEXT EDITION. USE THEM ONLY WITH CAUTION.

"HIGHER LEVEL PROGRAMMING"

Introduction to the use of the
Set-Theoretic programming language SETL.

by R. B. K. Dewar
E. Schonberg
and J. T. Schwartz

Table of Contents

Preface

The computer programming language SETL is a relatively new member of the so called 'very-high-level' class of languages, whose other well-known members are LISP, APL, and SNOBOL. These languages all aim to reduce the cost of programming, recognized today as a main obstacle to future progress in the computer field, by allowing direct manipulation of large composite objects, considerably more complex than the simple integers, strings, etc. available in such well-known 'middle level' languages as PASCAL PL/I, ALGOL, and Ada. For this purpose, LISP introduces structured lists as data objects, APL introduces vectors and matrices, and SETL introduces the objects characteristic for it, namely general finite sets and maps.

The direct availability of these abstract, composite objects, and of very powerful mathematical operations upon them, improves programmer speed and productivity significantly, and also enhances program clarity and readability The classroom consequence is that students, freed of some of the burden of petty programming detail, can advance their knowledge of significant algorithms and of broader strategic issues in program development more rapidly than with more conventional programming languages.

The price that very high level languages pay for their expressive power is a certain loss of efficiency. SETL should therefore be regarded, not as a tool for production-efficiency programming, but as a vehicle for rapid experimentation with algorithms and program design and as an ideal vehicle for writing 'one-shot' or infrequently used programs whose efficiency is of little consequence. It is also an effective tool for prototyping large systems purposes of design validation and early customer exposure, systems which if important enough can then be hand-translated into more efficient versions written in programming languages of medium or even low level. Experience with SETL will show that it is efficient enough for a surprising variety of purposes; nevertheless, it is still expensive to run, and will remain so until a new generation of high-performance microcomputers appear. In spite of this, SETL is a good vehicle for <u>discussing</u> program-efficiency issues, since it allows a graded approach to these issues, algorithm design being chosen first and data structures which realize them being chosen second, It will also be seen that the data structure representation sublanguage of SETL, described in Chapter X, is a powerful conceptual tool aiding such 'programming by stepwise refinement'.

Fairly polished versions of SETL are currently available on the DEC VAX and CDC Cyber, and less polished experimental versions on the IBM/370, DEC 10, and DEC 20. The systems running on all these machines are close to identical, all being produced from common system source by transporting an underlying systems-writing language from machine to machine. The relatively small differences between versions running on different machines (and under different operating systems on a given machine) are documented in Appendix X.

This book is intended for people who want to write programs in SETL. It does not assume knowledge of any other programming language, and is therefore suitable for use in an introductory course. We attempt to explain most of the mathematical concepts which play a role in SETL programs, almost all of which are in fact quite elementary. However, we do assume that the reader has a working knowledge of such basic concepts as set, sequence, etc.

The knowledge assumed is roughly equivalent to that which would be acquired in a good high school 'new mathematics' course, or in the first month of a freshman-level course in discrete mathematics.

We present considerably more material than can be covered in a one-semester introductory course. Chapter 1 provides an introduction to computer programming and an introductory overview of the SETL language. Chapter 2 introduces the major data objects of SETL, of which sets, maps, and tuples are most characteristic, and describes many of the language's operations. By The end of Chapter 2, the student is in position to write various interesting 'one-liners'. Chapter 3 then presents various basic control structure notions, qualifying the student to write interesting short programs. Chapter 4 introduces the most important control structures, namely PROCEDURES and their invocations.

Chapters 5 and 6 describe the remainder of the operations, expressions, and control forms of the language, except for backtracking, which is covered considerably later, in Chapter 8. Chapter 7 gives advice on program development, testing, and debugging, completing what can be considered the elementary part of the book.

The first seven chapters can be covered in a one semester introductory course, and can be skimmed rapidly by any reader reasonably familiar with at least one modern programming language, such as PASCAL, PL/1, ALGOL 60, ALGOL 68, or ADA.

The remaining chapters present more advanced material, which could be covered in a second programming course. Chapter 8 describes the I/O features of SETL systematically. Chapter 9 introduces the directory, program, module, and library mechanisms used to structure large programs. Chapter 10 presents SETL's data representation sublanguage and reviews various strategic considerations which play a role in data representation choice. Chapter 11 shows the language in action by presenting several more substantial applications of it, including a simple interactive editor and various computational geometry and graph algorithms.

SETL was developed at the Computer Science Department of New York University, by a group of which the present authors were members. The language has now been used by students in courses ranging from the introductory undergraduate to graduate courses in algorithm design. The style and order of presentation adopted in this book reflects some of the pedagogical experience gained in this way, especially at the undergraduate level.

Thanks are due to the many persons who helped to define and develop the SETL system. David Shields has been a mainstay throughout, inventing and implementing many system improvements, and developing documentation from which several of the sections of the book are drawn. Much of the first version of the system was written by Arthur Grand, and brought to solidity by Stefan Freudenberger. Thanks are also extended to Edith Deak, Micha Sharir, Robert Paige, Kurt Maly, Phillip Owens, Aaron Stein, Earle Draughon, Bernard Lang, Leonard Vanek, Steve Tihor, and Hank Warren, all of whom contributed to the development of the SETL system. Valuable design suggestions were contributed by our colleague Prof. Malcolm Harrison and gleaned from his elegant BALM language. Essential thanks are due to the

very helpful and hard-working group of summer interns who helped put this maunscript together and remove many of its errors during the summer of 1981: Leonid Fridman, Nathaniel Glasser, Barbara Okun, and Yi Tso. We also wish to extend thanks to Prof. Andrei Ershov and his group at Novosibirsk, who have aided the development and definition of the language from the very first days, Prof. Anthony McCann of Leeds University, and Drs. Su Bogong and Zhou Zhiying of Tsinghua University, whose more recent involvement has been most valuable.

Finally, thanks are due to the research administrators who fostered the development of SETL through its early, relatively isolated years. Among these we should particularly like to thank Milton Rose, who launched our development effort during his years at NSF, also Kent Curtis and Tom Keenan of NSF, who fostered it during the period in which the NYU group was struggling toward a reliable and acceptably efficient implementation. We hope the success of the system will justify their confidence.

CHAPTER 1

PROGRAMMING CONCEPTS.

Chapter Table Of Contents

1.1 Programs

To program is to instruct a computer to perform certain desired actions. For example, using the programming language to be described in this book, you can write the instructions

```
print(54 + 45);
print('The difference of twelve and nine is:', 12-9);
print(55*55);
```

and submit them to a computer. Then, if the instructions have been properly typed and submitted, the computer, after first digesting them, will obediently produce the following results;

```
99
The difference of twelve and nine is:  3
3025
```

The instructions you submit to the computer are known as source code or input; the results which it prints are known as output. Programming is therefore the art of devising inputs which describe the output that you want to produce.

This first example suggests that programs can only deal with simple numerical quantities and can only describe simple arithmetic calculations. This is by no means the case. Computers are not just numerical calculating machines, they are general information-processing engines and can manipulate information of arbitrary structure and complexity. This basic fact will be strongly emphasized by the programming language, SETL, described in this book. For example, you will see that it is easy to manipulate arbitrarily

complex tables, for example, tables of names, addresses, telephone numbers, birth dates, and salaries having the form

```
[['Aldo Gonzalez','45 Ellwood Ave','278-3591','12-12-45',21315],
  ['Jimmy Archibald','1315 Bole St','479-1919','5-31-78',0],
  ['Willa Cross','111 Mocking Pl.','275-1212','7-19-00',6700],
```

Such tables can be built up, sorted, searched for particular elements or combinations of elements, extracts and statistical summaries of them can be prepared and printed, etc. All this will be easy to do once you have learned the programming language described in this book, which can handle a table like that shown above just about as comfortably as it can handle a simple number like 23.

However, although the programming language to be presented is powerful, and although computers are extremely fast and perfectly accurate, they are also unintelligently pedantic and narrowly literal in their reactions to the instructions which they receive. This has two fundamental consequences, which you as programmer (i.e. as a would-be author of programs), must always keep in mind.

(i) The computer will always do exactly what it is instructed to do, neither more nor less, and will do this if its instructions are legal, irrespective of whether these instructions are reasonable or unreasonable from some larger point of view. This can be quite disconcerting at first, since it can easily lead to unexpected consequences. When you ask a person wearing a watch 'Can you tell me what time it is?', you expect an answer like 'It's 3:15'. A person acting like a computer would instead answer, 'Yes, I can'; but would never actually tell you the time until you actually uttered a direct and unambiguous command like 'Tell me the time'. Therefore hints, even hints that a person might regard as utterly clear, are quite useless to a computer. It does not know, or care, what you have in mind for it to do: it only knows what it has been directly and unambiguously commanded to do.

This it will do with perfect fidelity, even in circumstances in which even a boundlessly faithful and determined person would realize that something is quite wrong with the instructions given him and would ask for more reasonable instructions. For example (though they would have to be expressed a bit differently), the following instructions can be given to a computer, and would then be followed literally:

```
instruction 1:  print ('Hello There');
instruction 2:  go back to instruction 1.
```

Given these instructions, the computer will, like a phonograph stuck on a groove in a cracked record produce the output

```
Hello there
Hello there
Hello there
Hello there
Hello there
Hello there
```

...

line after line, thousands, millions, or even billions of times, as long as paper remains in its printer, power continues flowing to its circuits, the building which houses it has not burned down or collapsed in an earthquake, and as long as neither the human operator (nor the automatic operating system) which regulates it have grown suspicious enough to switch the computer to another activity. In such circumstances, the exhilarating magic of the computer sours after the fashion of the well-known tale of the sorcercer's apprentice.

(ii) In order to follow instructions given to it, the computer must first be able to digest and understand them. The linguistic abilities of computers are limited, and their abilities to recover from errors are also limited. Hence the approach it takes is extremely pedantic. In particular, you will find that it insists that commands submitted to it must adhere precisely to the grammatical rules, and even to the 'petty rules of punctuation, of the programming language which it is set up to handle. The omission of so much as a dot, the misspelling of a single name or a single command keyword, the substitution of a single colon where a semicolon is wanted, the insertion of a single blank space where it is not wanted: all these petty errors are fatal, and will cause the computer to reject a set of instructions before it even attempts to follow them.

So, for example, the three commands appearing at the very beginning of this section would not be executed if they were submitted as they stand, but only if they were preceded by a required line serving to introduce and name them, and followed by a required 'trailer' line, thusly:

```
PROGRAM sample_program_number_1;
print(54 + 45);
print('The difference of twelve and nine is:', 12 - 9);
print(55*55);
END;
```

Packaged in this way, our three original commands come to constitute a complete and valid program in the SETL language, acceptable as it stands.

The difficulty that computers have in coping adequately with error causes them to react to tiny program details in a pedantic manner, to which the beginning programmer must grow accustomed. If, for example, the program shown above is submitted as

```
PROGRAM sample_program_number_1;
print(54 + 45),
primt('The difference of twelve and nine is:' 12 - 9);
print(55 * 55);
END;
```

it will be rejected without producing any output. In fact, three errors, each fatal in spite of the fact that it can easily be corrected (and, indeed, might never even be noticed) by the human reader of these instructions, occur in the text shown above. These sins, damning in the computer's view though trivial to the human viewer, are:

(a) Substitution of a comma for a semicolon in the second line.

(b) Omission of a comma after the terminating quote in line 3.

(c) Misspelling of print as primt, also in line 3.

Clearly, then, to interact in a satisfactory way with a computer you will have to come some distance toward compromise with what will at times seem like a maddeningly literal, detail oriented, robot mind. But these initial irritations can be overcome, and, once you have overcome then, you will find the amazing powers of an infinitely flexible machine at your command.

As a programmer, you will find it instructive to realise that programs existed long before computers were invented, even though computers have given them forms different from what they had before and hav vastly extended their scope. Mankind first encountered programs early in the new stone age (or perhaps even in the old stone age), when basket-weaving and palm-leaf weaving were invented. Basket patterns, palm-leaf weaves, rug patterns, knitting patterns and musical scores are all programs, that is, are sequences of instructions involving choice and repetition, whose execution produces outputs which are larger and more interesting than the sets of instructions from which they were produced. All of these activities exhibit the elements of repetition and choice (i.e., repetition with planned variations) which is so characteristic of programming, and which we will constantly meet with in this book. Note in particular that

(a) In programming, as in knitting, it may be necessary to execute an instruction, or a group of instructions, more than once. Most programs will therefore involve repetitions, or even repetitions within repetitions, as in 'knit three stitches and then purl two, repeating twenty times for each row, for ten successive rows; then knit five rows of 100 stitches each.' The number of repetitions desired can be specified either by an explicit count, as in the preceding example, or by stating a condition which depends on the state produced by prior repetitions (as in cooking: 'beat steadily until the mixture thickens'). Both forms of repetition will be encountered again and again in the chapters which follow. Since computers execute more than a million elementary instructions per second, computer programs are even more dependent on repetition than knitting and weaving patterns are: a repetition-free program would run for no more than a tiny fraction of a second.

(b) Simple repetition, like the endless repetition of a single stitch in knitting, can only produce an unending featureless cloth. To produce something more interesting depends, in programming as in knitting, on repetition with variation, and on proper combination of repetition with choice, like the choice which appears in the example 'If a size 25 sweater is desired, repeat for 30 rows, but if a size 27 sweater is desired, repeat for 36 rows'. The fact that conditional instructions of this kind can be used in a program makes it possible to produce a wide variety of outputs and write programs that can be used in an immense variety of circumstances. In fact, complex sequences of choices are much more characteristic of programs than of any other kind of plan, pattern, instruction, or recipe, since the extreme accuracy of the computer makes it possible to plan and follow long sequences of choices and variations that would soon leave a person trying to

carry them out exhausted and hopelessly confused.

(c) Programs, like knitting instructions, are relatively unchanging objects; but their execution, like knitting, is a dynamic activity. A program is no more the same thing as its output than a set of knitting instructions are the same thing as the sweater they describe; nor should we confuse a program with the computer on which it runs, any more than we would confuse a set of knitting instructions with the needles used to execute these instructions.

1.2 An informal overview of SETL.

The programing language SETL has many powerful features, and it will take well over a hundred pages to explain them all. Therefore this short section can only give you a glimpse of some of these features. Nevertheless, before we march forth to explore the terrain systematically, it is worth previewing SETL's most characteristic features informally. For this purpose, we consider a simple example. As its name implies, SETL makes it easy to work directly with sets. Suppose therefore that the following

set of numbers is given:

(1) {13, 11, 45, 0, -16, 21, 85, 46, 80}

and call it s. The problem we wish to consider is that of finding the median of the numbers in s, namely the number which would come halfway beween the first and the last element of s if the elements of s were arranged in ascending sequence from lowest to highest, namely as

(2) [-16, 0, 11, 13, 21, 45, 46, 80, 85]

(In our example, this median is clearly 21). If (as in our example) s has an odd number of elements, then the median (which is often used in statistics to represent a 'typical' member of a set s) can be defined as follows: it is the unique element x of s such that there are as many elements of s which are smaller than x as there are elements of s which are larger than x. If s has an even number of elements there are (as it would if we dropped the number 85 from our example) nothing lies exactly in the middle, and we could argue about which of the two numbers (e.g. 13 or 21) lying equally close to the middle of an ordered sequence like (2) should be considered the median. To avoid this complication let us agree for the moment that we will only consider sets having an odd number of members. For such sets, the median is simply the number x defined by the following condition:

(*) The number of members of s which are less than x is equal to the number of members of s which are greater than x.

In SETL, a set like (1) can be read in (for example, from the keyboard of a computer terminal, or from a punched card), simply by writing the command

```
READ(s) ;
```

Once having read s in, we may want to find, and print, its median. As with all programming tasks, this can be done in several different ways. If we knew how to arrange the elements of s in order, we could simply find this arrangement, take the element which comes in the middle, and print it out. Arranging elements in order is called sorting; we will study many techniques for sorting later in this book, and any one of them would put us into position to use this approach to findng the median. However, it still is too early to show you how to do anything quite this complex, and hence we shall follow another path, namely we will use the definition (*) directly.

In order to do this, we must first be able to form 'the number of members y of s which are less than x'. Since SETL makes it easy for us to form sets, and allows us to get the number of elements in any set t simply by writing #t, this is easy: we simply form the set of all members y of s which are less than x, and then take its number. The set we want can be formed simply by writing

(3) {y IN s | y < x } .

and its number of elements is therefore

(4) #{y IN s | y < x}

Similarly, the number of elements in s which are greater than x can be written as

(5) #{y IN s | y > x}

Concerning the construct (3), which is known in SETL as a set former, we can make the following remarks:

(a) It is written in a fairly standard mathematical notation, which will be familiar to anyone who has studied much mathematics (even grade-school or high-school level 'New Math').

(b) The notation (3) should be read as follows:

(b.i) The curly brackets surrounding the rest of formula (3), which are sometimes called 'set brackets', are simply read as 'the set of'.

(b.ii) The next part, i.e. y IN s, is read more or less as it stands, i.e. as 'y in s', or perhaps as 'all y in s', thus giving 'the set of all y in s'.

(b.iii) The '|' symbol is shorthand for 'such that'.

(b.iv) The condition following | is standard mathematical notation which is read as it stands, giving altogether

'the set of all y in s such that y is less than x'

as the English reading of (3), and similarly

'the number of elements in the set of all y in s such that y is less than x'

and

'the number of elements in the set of all y in s such that y is greater than x

as the readings of (4) and (5) respectively.

We can therefore express the condition (*) which defines the median simply by writing

(6) #{y IN s | y < x }= #{y IN s | y > x}

There will exist such an x if and only if the number of elements in s is odd. SETL allows one to test for existence of an x satisfying the condition (6), and to find it if it exists, simply by writing

```
(7) EXISTS x IN s | #{y IN s | y < x}= #{y in s | y > x}
```

which in English reads roughly

```
'there exists an element x in s such that the
      number of elements in s which are less than x equals the number
         of elements in s which are greater than x'
```

(Note that the first | in (7), like the others, can be read as 'such that'.) If the median exists, i.e. if the number of elements in s is odd, we want to print it out; otherwise, only a message announcing that s has an even number of elements will be printed. This sort of conditional action, determined by a condition which cannot be evaluated until actual data has been read and examined, is expressed in SETL (as in most other modern programming languages) by an 'IF statement'. A full account of this important command will be given in Chapter III; however, even without this full account, the meaning of the following IF-statement, which does what needs doing in the present case, should be clear:

```
(8) IF EXISTS x IN s #{y in s | y < x }=#{y in s | y > x} THEN
        PRINT('The median is:', x);
    ELSE
        PRINT('No median, the set s has an even number of elements.');
    END;
```

Note the following details concerning the command (8):

(i) To produce output printed on paper or displayed on a terminal, the PRINT command is used. This can either print a simple message (like the second of the two PRINT commands shown above, or (like the first PRINT command) can be used to print both a message and a quantity that has been calculated elsewhere in the same program (like the -x- in example (8)).

(ii) The IF-statement appearing in (8) must be terminated by an occurrence of the word END, which is needed to mark the end of the IF-statement unambiguously.

(iii) The rules of SETL punctuation require both the PRINT commands appearing in the above example, and also the whole IF-statement, to be terminated with a semicolon.

As was already noted in Section 1.1, both an introductory 'header line' and a terminating 'trailer line' must be added to (8) before it can be run. Adding these lines, we arrive at the following fully set-up program, which can be used to read any set s of integers, and print out the median of s if s has an odd number of members:

```
PROGRAM find_the_median;
READ(s);
IF EXISTS x IN s | #{y IN s | y < x} = #{y in s | y > x} THEN
      PRINT('The median is:', x);
```

```
ELSE
      PRINT('No median, the set s has an even number of elements');
END;
END PROGRAM find_the_median;
```

Though simple, this program illustrates several of the most significant features of the SETL language: SETL allows us to define, construct, compare and in general manipulate sets of values ; such sets can be searched to find whether elements exist that satisfy a given property; such sets can also be read and written, and (as we shall see in Ch.2) modified in a number of ways. We shall see, as our study of the language progresses, that sets and set operations are particularly versatile concepts for problem-solving and programming, and that SETL allows its skilled user to solve complex problems with greater ease than that afforded by most other programming languages.

1.3 The steps of programming; how to run your program and read its results.

Knitting instructions, basket weaving patterns, recipes, even weaving instructions for handlooms; all are intended to be executed manually by a person, who can at least be trusted to stop if he starts to get into trouble because something is wrong with the instructions. However, programs, like weaving set-ups for large automatic looms, will be executed at high speed by a machine. If this is not to lead immediately to failure, or still worse to a high-speed outpouring of trash, programs must be planned, set up, and tested carefully before they are released for full scale high-speed execution. This involves a whole series of steps:

(I) One starts with an initial conception: what would be interesting, useful, scientifically or commercially valuable, to have? The answers to such questions come from outside the technical field of programming.

(II) Once a goal has been formulated, what patterns of repetition and choice, what ingenious shortcuts, allow the desired output to be produced most simply and efficiently? These questions touch upon an area of program and algorithm design that lies outside the scope of this introductory book; however, the many programs presented in the chapters which follow will illustrate some of the numerous techniques for clear and effective design that are available to the knowledgeable programmer.

(III) Once both a goal and a general plan for realizing it have been specified, there begins the detailed work of restating the plan in terms of the specific toolkit of instructions made available by the programming language that one is using. This is the labor of programming *per se*. As will be seen, the SETL language presented in this book supplies its user with very powerful tools of expression, and therefore allows programs to be expressed more easily, simply, and directly than they would be in other, less abstract programming languages. But these tools must be learned carefully and then used accurately: computers enforce a compulsive attention to detail that takes some getting used to. If used accurately, they will allow you to write both short programs, like the examples shown in the preceding section, and sophisticated programs many hundreds or thousands of lines long which realize very complex functions.

(IV) After being typed at a terminal or punched on cards, a program can be passed to the computer on which it is to run. This will trigger a whole sequence of behind-the-scenes activities, with which you will only be peripherally involved, but of which it is important to have some understanding in order to cope with the various things that can go wrong between the time that your program is first entered into the computer and the time, several seconds to several hours later, when output finally emerges. Though differing somewhat from machine to machine, these steps will generally be more or less as follows:

(i) Your program is passed, as a passive file of *data*, to another group of programs, pre-stored in the computer. These programs, which collectively comprise the computer's *operating system*, share the computer's power among the many users entering jobs at card-readers and terminals, all of whom require, and will eventually get, a quantum of service from the computer system.

The first thing that the operating system programs do is validate your identity as an enrolled user of the computer. If this check fails, you will be refused service. This will happen immediately if you are using a terminal and fail to identify yourself to the operating system's satisfaction. On the other hand, a 'batch' or 'card oriented' system normally looks for user identification on the first card of each deck submitted to it; this identification usually consists of a user name, password, and perhaps a few other information items such as the account to which the cost for a program run are to be charged. If any of this information is invalid, the computer system rejects your job and only very enigmatic information, for example a single sheet bearing your name and a cryptic refusal to service your progam, is produced. However, if you pass the operating system's user validation check, the program you have submitted will be entered onto a pending work queue, where it will wait, along with work entered by other legitimate system user to be scheduled for future attention by the operating system. In an interactive system run from a terminal this should normally take no more than a few seconds; in a card-oriented batch system it can wait anywhere from moments to hours.

(ii) When your turn to be served further comes up, the first line or few lines of information supplied with your program are examined by the operating system programs running on the computer. These first lines, known As command lines (or perhaps as control cards, job control cards, or JCL) serve to tell the operating system, which provides many services to many users and deals with many programming languages, which one of its services you want.

To run a program in SETL, your command line or lines will have to convey the following information to the operating system:

(1) The Language to be used (i.e., SETL).

(2) The location of the SETL program to be processed. In a batch system this will generally be a deck of punched cards following immediately after the comma nd lines, but in an interactive system it will more often be a file of lines which you have previously entered into the computer using an auxiliary 'editor' program. In the latter case, the name of this pre-stored program file must be indicated.

(3) The location of any input data which your SETL program needs to read. In batch systems, the data may simply follow the text of your program, in the same deck of punched cards. In terminal systems, this data can either be obtained from a pre-stored file or read directly from your terminal, in which case you will have to type it in, in response to queries which your program sends to the terminal as it runs.

(4) The destination to which output produced by your program is to be sent. In batch runs, this will be a 'temporary file' which is printed after your program has halted (or, in the event of trouble, after your program has been halted forcibly by the operating system). In interactive runs, your output either will be written to a file which you can examine after it has been produced, or will be sent to the screen of your terminal, in which case output will appear as your program runs.

(5) You can supply additional information to select 'options' which influence details of your run. Descriptive material concerning these options is found in section 8.2.

If any of the information contained in the command line which initiates execution of your program is defective, the result will be almost as catastrophic as if your user validation had failed. Your program will not run, and your only output (in a batch sytem) will be a page or two of information recording the fact that your command lines contained an error. To get past this barrier you must repair your command lines, entering them in completely error-free form. Make sure you understand all details of the required form for these few (but operating system dependent) lines; consult an expert immediately if trouble persists.

(iii) Assuming now that both your identification and your command lines have proper form, the operating system programs will prepare for the processing and execution of the program which you have supplied. Though this involves many detailed steps, some of which are described below, the two basic things that the operating system needs to do are just the following:

(1) The program which you have supplied will be examined, checked for exact conformity to the rules of the SETL language, and, if it passes this check, translated into an internal program form with which the computer can work directly. This first step, checking and translation, is called compilation, and the program which carries it out is called the SETL compiler. (Note that compilation is necessary because the form of SETL which you write and submit to the computer is designed for human, not for machine, convenience; it must be translated into a more machine-convenient form before your program can actually be run.)

(2) After translation into appropriate internal form, the instructions give in your program are actually performed, (possibly) producing output. This step is called execution, and the program which carries it out is called the SETL run-time system.

(iv) Errors can, and often will, be detected during either of the two preceding steps. Grammatical and other relatively 'gross' errors in the use of the SETL language will be detected during compilation. Unless you have switched off the 'listing' option of the compiler, it will print out and number all the lines of your program exactly as it sees them, and if it detects any grammatical errors it will flag them in the resulting 'listing' of your program, which forms the first part of the output which you receive.

If compilation errors (also called syntax errors) occur in your program, then, as indicated by a message 'ABNORMAL TERMINATION' which will appear in the above processing of your program will end as soon as the compiler finishes its work; your program will not actually be run. To get further, you must correct all grammatical errors. Once this is done, all diagnostic messages will disappear, the first part of your compilation listing will appear as follows, and your program will move on, passing, as one says, into execution.

Substantially later in your listing there will appear the output which your program has produced. The three lines of output produced by the sample program we have been considering would look like this:

```
99
The difference of twelve and nine is: 3
3025
```

In looking for this output in your listing it is important to realize that the output is actually preceded by several dozen more lines of standard 'boiler Plate' which you will grow accustomed to seeing in your output listings and can normally scan over quite rapidly. This additional material appears because the SETL compiler is a large and complex program which actually operates in three phases:

(1) A 'parse' or 'grammatical analysis' phase, which analyzes your program, checking it for syntactic validity and breaking it down into the elementary clauses of which it is composed. This produces the section of the listing, headed CIMS.SETL.PRS, which is shown X pages above.

(2) A 'semantic analysis' phase SEM which takes the collection of elementary clauses produced by the PRS phase, applies additional validity checks to then, and continues the process of transforming your program into an internal form which can be interpreted directly by the computer.

(3) A 'code generation' phase COD, which completes the translation process begun by SEM.

See Appendix 1.7 for a description of the standard boilerplate which the SEM and COD compiler phases put into your compilation listing.

Other common kinds of error.

Once the PRS, SEM, and COD phases of the SETL compiler have successfully translated your program into its internal form, it is passed, in this form, to the so called SETL 'run-time' or 'execution' system, which then attempts to follow these translated instructions literally. (The translated form of your program is logically equivalent to the program which you have supplied, but is recast into a form that the run-time system can work with more easily.

However, several further sources of error can still give your output an appearance totally different from what you expect.

(a) You may have misunderstood what your program is really saying. For example, you may not have realized that suitably placed 'print' commands are necessary if any output is to be produced, and may have imagined that results are printed merely by virtue of being calculated by your program. In this case, no output at all may appear.

An endless variety of other small logical errors of this sort are possible, and only experience will teach you how to avoid them. Removing errors of this sort is called debugging; hints concerning debugging techniques are found in Section 7.2, 7.3.

(b) Attempts to execute illegal operations are another common consequence of misunderstanding what a program is really saying. Suppose, for example, that your program contains the command

```
print(x + y) ;
```

but that prior instructions have given x the integer value 1 but not defined The value of the variable y. Addition of an integer and an undefined value is illegal, and the SETL run-time system will detect this violation when it attempts to evaluate x + y. The run-time system will then generate a so called run-time or execution error, and program execution will be terminated immediately (aborted). In such case , your output will end with a run-time error message, describing the problem encountered. When this happens, you may want to rerun the program, using some of the additional debugging options described in Section 8.5.1, to gather additional information about the location and cause of the error.

(c) If the logic of your program is in some way faulty, your program may not reach its termination, but may instead loop endlessly, in which case it can either produce output forever, or produce no output at all. (The hypothetical program

```
instruction 1:  print ('Hello there')
instruction 2:  go back to instruction 1
```

illustrates the first of these possibilities.) If your program starts to loop, then the operating systems programs (which always, so to speak, lurk in the background, checking on what other programs are doing) will eventually detect the fact that your program is producing an illegally large volume of output or that it has outrun the time quota which the operating system established for it. When this happens, your program will be forcibly

terminated by the operating systems programs, which will write a message explaining what has happened.

You will need to grow familiar with the appearance that your output listing takes on when these various common problems are encountered. Here, for example, is the output that results from mistyping the number '45' in the second line of our sample program as 'x5', in which case it will be interpreted as the name of a variable, which the run-time system will find does not have any assigned value.

```
*** ERROR AT STATEMENT 2 IN ROUTINE S$MAIN
    INCOMPATIBLE TYPES FOR -A- AND -B- IN -A + B-.
```

Note that this message identifies the offending statement, by number, as 'statement 2' of your 'main' program (in this simple case, all that exists is a 'main' program ; in the more complex cases which we will begin to introduce in Chapter IV, both A 'main program' and numerous 'subprocedures' can exist). Beyond this rather terse statement, no other information is given (however, more information can be produced using the debugging options described in Section 8.5.1.)

1.4 Advice to the would-be programmer

As will be seen, the SETL language presented in this book furnishes you with many very powerful tools, and also makes it possible to create new tools by combining more elementary ones into procedures which you yourself can define. Nevertheless, it provides only certain specific facilities, and not, in some magical way, everything that you might want, think it would be convenient to have, or even imagine to be available. You will therefore have to distinguish carefully between the facilities which the language makes available and those which it does not, learning the nature, form, and especially the purpose of every feature and facility of the language, but also learning what it does not make available directly (especially if this is something you would like to have and wish it did make available directly). It is as senseless to plan programs that make use of nonexistent programming language features as it is to work out seven-color, three-hundred thread patterns for an automatic knitting machine that only allows four colors and 180 separate threads.

Of course, since the computer is infinitely more flexible than any other kind of machine, it is likely that you can find a way of building up any well-defined facility which you can conceive clearly and describe precisely. However, this can only be done by accurate use of the facilities of the language (SETL) that you will be using, not by imagining that you can suddenly leap out of its confines. Thus, even to go successfully beyond what is originally present in the language you will have to learn to distinguish accurately between the tools it provides and those which it omits.

Here, an important psychological point needs to be made. To accomplish an operation which some feature of a programming language provides for directly is easy, provided that one recalls the feature and can look up whichever of its details are relevant. But this kind of memorization merely skims the surface of programming. An infinite variety of more complex and interesting operations can also be programmed, but to do so one needs to decompose them into more elementary operations which can be carried out more directly, and so on through progressive stages of decomposition, until one reaches operations which can be expressed directly by single comands of the programming language with which one is working. Though helpful hints about how to do this can and will be given, this process of decomposition cannot be accomplished by application of any simple recipe, it requires problem solving and invention. Now, unless at some time in the past you have been either a devoted and successful puzzle enthusiast, chess, bridge or checker player, or a mathematics student, you will probably find that programing makes unexpectedly strenuous demands on your problem-solving muscles, demands for which your past education has probably given you very little preparation. Indeed, with few exceptions, school courses teach memorization, or at best application of memorized procedures, but not true, no-holds barred problem solving of the kind you will encounter in learning to program. In History you have learned facts and interpretations, in Chemistry more facts, in undergraduate Physics you have learned formulas and how to apply them; in mathematics, up to and well into calculus, you have also memorized various procedures and how to apply them. Therefore it may very well be that in becoming a programmer you will have to master the intellectual art of problem solving for the first time. The following

paternal remarks are intended to help you cope with this challenge.

(a) Don't panic. Although some people are better at problem solving than others, the ability to solve problems, like the ability to cook a good spaghetti sauce or dance the waltz acceptably, can be learned by anyone. Don't let your instructor's problem-solving speed intimidate you. He probably has both talent and years of experience; Of course you will need time to catch up with him.

(b) On first facing a problem that you have never solved before, you will feel confused. Again, don't panic. Remember that you are not trying to remember a fact which you have forgotten, rather, you are trying to search out, to devise, to discover something which, for you, is new. The initial confusion (which everybody, even the strongest problem solver, is bound to feel at first) is not the end of your efforts to solve the problem: it is merely the start of the beginning. Don't say to yourself: 'I don't see the answer; I am confused; I give up'. Instead, say 'I am in process of wrestling with, and dispelling, the initial confusion which every new problem generates,' and fight on. Significant problems, like nuts, have hard shells, and can only be cracked if they are examined closely enough for their lines of cleavage to be found. Pick the problem up, attach yourself to it, and begin to turn it over, searching from all angles for the hints which will unlock it.

(c) Explore the leads which occur to you, combining caution and boldness. Can you see a fragment of the solution? Can you guess one command which will be helpful? Can you solve any part of the problem? Can you see any way of breaking the problem into two or more parts which look easier to solve than the whole problem does? If you have solved some part of the problem, what problem remains? Can you see any way of extending your partial solution to cover more of the problem? If you can't solve the or iginal problem, can you solve some easier problem that has significant similarities to it? If so, can this solution be improved enough to solve the original problem, or at least a problem substantially closer to it? If not, what is the easiest similar problem which you cannot solve? Why not? What feature of it prevents solution? What, if anything, can be done about this feature?

(d) Don't give up too easily. Remember that a programming problem, like a jigsaw puzzle, may have to be solved one piece and one clue at a time.

(e) If no progress seems to be possible along a given line of attack, try to find another approach. Sleep on the problem and start afresh with a new approach the next morning.

(f) If a problem seems intractable, go to an appropriate book and look up a solution, or to a helpful, more knowledgeable person and have the solution explained. But take this help actively, not passively. Ask yourself: What is the key trick that I failed to discover? In what other situations can this new trick be useful? What part of the problem could I have solved with what I knew before; what aspect really requires the new method that has just been explained? Practice using the new method on a few simple examples you make up for yourself, and ponder it carefully, to make sure you digest it.

(g) Accustom yourself to dealing with concepts and methods, not with memorized program fragments. Although memorized fragments, like memorized sequences of chess moves, are useful, and even though the experienced programmer may have memorized dozens or hundreds of them, no two situations are exactly the same in programing, any more than they are in chess. Your basic need in learning to program is not to remember programs presented in a book and adapt them slightly to new situations: it is to learn how to invent general logical plans, and to master the principles which will allow you to do this, along with the language in which you will have to explain your plans to a computer. General methods, principles, and approaches will retain their usefulness over a wide range of circumstances, while ill-conceived attempts to adapt a textbook example to do something it Was not designed to do will often be less profitable than wiping the slate clean and starting afresh.

(h) Train yourself to accuracy, but don't be overly afraid of errors. Computers have only a limited capacity to deal sensibly with errors. On the other hand they are infinitely patient, and will give you all the chances you need to remove the errors initially present in your program. Because of the high degree of accuracy with which programs need to be prepared, errors are as omnipresent in programming as in clutter in kitchens and sawdust in woodshops. Remember that no one is looking impatiently over your shoulder as you develop a program; you can have all the tries you want, and only your final success counts. The computer is infinitely patient; one must scribble to write; everything along the way to final success is just scrap-paper to be thrown out.

Your aim in dealing with errors should not be to avoid them fearfully, but to learn to recognize them clearly, understand the violations of rule and principle which let them creep in, and remove them swiftly. As long as your programs are moving rapidly toward correctness, errors are tolerable. Only errors which you cannot recognize and do not know how to remove need to be considered major problems.

(i) On the other hand, accumulation of numerous unnecessary errors through gross carelessness or misunderstanding will wind up wasting large amounts of your time as you struggle to remove mistakes that a little more care could simply have avoided. Hence it is really important to train yourself in accuracy, and to learn to use the programming language to be presented in this book cleanly and grammatically. You will want to study it closely, learning its facilities, restrictions, style, and inner rhythms. As your programs evolve toward completion, you will want to review them carefully and suspiciously, trying to search out all errors in programming language use or in underlying logic; all hidden defects which might force you to waste time later. As we have said, the programming language to be presented in this book is a kit of powerful tools for your use: you will want to inspect all the tools in this kit, and to understand and reflect upon their capabilities, restrictions, and intended use. This will help you to develop into a skilled practitioner able to do everything in the clearest, most direct, most effortless way.

1.5 How to type a program; Character sets

If the terminal or keypunch machine with which you are working has all the characters which appear in SETL programs in this book, then you can type your programs exactly as this book will show them. The special punctuation characters required are

<	less than
>	greater than
=	equals sign
(	left parenthesis
)	right parenthesis
'	quote mark (apostrophe)
.	period
,	comma
:	colon
;	semicolon
/	slash
+	plus sign
-	minus sign
$	dollar sign
?	question mark
#	number sign
_	underline
{	left set bracket
{	right set bracket
[	left square bracket
]	right square bracket
|	such that symbol

When not all these characters are available, standard substitutions can be used for some of them. These include the following

{	can be written as	<<
{	can be written as	>>
[	can be written as	(/
]	can be written as	/)
|	can be writen as	ST

The remaining characters are replaced if necessary by single characters which type differently. For lists of these character substitutions, you will have to consult implementation specific information available from the computer center in which your programs are being run.

Some, but not all, implementations will make both upper case (capital) and lower case (small) versions of all the alphabetic characters available. When this is so, programs can be typed either in capital letters, small letters, or any helpful and pleasing combination of the two. For example, the command

```
print(3+5);
```

can also be typed as

```
PRINT(3+5);
```

or as

```
Print(3+5);
```

or even as

```
PrInT(3+5);
```

The SETL system always transposes all 'keywords' like PRINT appearing in a program into upper case, and works internally with these upper case versions. Only characters appearing within quotation marks (i.e. in 'quoted strings', see Section 2.1) are retained in their original lower case forms. This means, for example, that the statement

```
PRINT('hello there') ;
```

will produce the output

```
hello there
```

whereas the statement

```
print('Hello There') ;
```

will produce the output

```
Hello There
```

Extra blanks are generally ignored, and can therefore be used to space out your program text to make it more readable. For example,

```
print(3+5)
print(3   +   5)
print(3     +5     )
```

will all produce the same output, namely

```
8
```

The only places in which blanks are forbidden to appear (or have meaning if they do appear) are within constants, standard keywords, and variable names. For example,

print	cannot be written as	p rint
1000000	"	1 000 000
counter_1	"	count er_1 .

SETL instructions are terminated by semicolons, and can be continued over as many lines as necessary. This means that the instruction

```
print(3+5);
```

could also be typed as

```
print
     (
      3
       +
5);
```

if there were any sensible reason for doing so. See Section XXX for rules concerning the continuation of a quoted string from one line to the next.

The dollar sign '$' is used to indicate the points at which there begin explanatory 'comments' that are intended to be helpful to a programs' human reader but which are ignored by the SETL compiler. See Section XXX for a

discussion of the use of comments in programs, a very important subject.

1.6 Exercises

1. Find out how to run the program shown in Section 1 on your computer, and run it.

2. How could you define the median of a set having an even number of integer members? Can you modify the program shown in Section 3, so as to make it work irrespectively of whether the set of integers supplied to it has an even or odd number of members?

3. Take the median-finding program of Sec.1.3, and introduce various typing errors in it. Submit these mangled programs to the SETL compiler, and study the resulting error messages. Try to predict what the response of the compiler will be to each error you insert.

4. Jot down a personal inventory of your own history as a problem solver, listing all your experience in such relevant activities as mathematics and science classes (especially solution of 'original' programs), chess, bridge, crossword puzzles, jigsaw puzzles, recreational puzzles, etc. Do you feel that you have quite considerable experience as a problem solver, Or only a little?

1.7 Appendix: More on how to read your output listing.

Here, for example, is how the compilation listing of the sample program shown in the preceding section would look if it contained two small grammatical errors, namely omission of the comma shown in the third line of the program (see p. XXX) and replacement by a colon of the semicolon which should end its fourth line:

```
PROGRAM sample_program_number_1;
1    1      PROGRAM sample_program_number_1;
2    2      print(54 + 45);
3    3      print('The difference of twelve and nine is:'  12 - 9);
******** ERROR 3: EXPECT RIGHT PARENTHESIS
   PARSING:  45 ) ; PRINT ( 'The difference of twelve and nine is:' 12
                           ----------------------------------------
4    3      print(55*55):
******** ERROR 9: EXPECT ASSIGNMENT OPERATOR
   PARSING:  'The difference of twelve and nine is:' 12 - 9 ) ; PRINT
                                                              =------
5    3      END;
******** ERROR 91: EXPECT VALID STATEMENT
         PARSING:  55 ) : END ; ;
                             =
*** COMPILATION TERMINATED BY UNEXPECTED END-OF-FILE ***
         PARSING:  ) : END ; ; ;
                           =

  NUMBER OF ERRORS DETECTED = 3
ABNORMAL TERMINATION.
```

Note the following concerning this 'compilation listing'

(1) The compiler numbers the lines of your program. Lines are numbered sequentially down the left of the listing. (The compiler inserts these numbers to make lines easier to refer to. Do not type in these numbers yourself.)

(2) Just to the right of these 'primary line numbers', there appear other, similar but slightly different, 'secondary line numbers'. These secondary line numbers are needed primarily for longer programs consisting of multiple procedures (see Chapter IV), to allow line numbering to be restarted at the beginning of each procedure. (Again, do not type in these numbers yourself, the compiler will insert them.)

(3) Following these numbers, the appropriate line of your program appears. These lines constitute the definitive version of your program, as it has actually been seen by the compiler. Check them carefully. If they differ in any way from what you think you have typed, then a typing error has occurred; this must be fixed before you can go any further.

(4) Immediately following each line in which the compiler has detected (or thinks it has detected) an error, there appears a so-called diagnostic message, flagged with 8 stars and the word ERROR, as in

```
******** ERROR 3: EXPECT RIGHT PARENTHESIS
```

After each such line, there appears a second diagnostic line, starting with the capitalized word PARSING, as in

```
PARSING: 45 ) ; PRINT ( 'The difference of twelve and nine is:' 12
```

Parts of this latter line will be underlined, in part with dashes '-', in part with equal signs '='.

The diagnostic or ERROR message that the compiler supplies when it detects or thinks that it has detected an error consists of an error number (-3- in the example given above) and a short statement (in our example, 'EXPECT RIGHT PARENTHESIS') representing the compiler's guess as to what the error was. Concerning this, you must be aware that, while very accurate in its treatment of error-free programs, the compiler has a very limited ability to deal accurately with errors, and that these statements, which represent rather nearsighted guesses only, are frequently wide of the mark. In the above example, the compiler guessed (wrongly) that you meant to end the print statement immediately after the first message, i.e. that what you meant to type was

```
print('The difference of twelve and nine is:');
```

Making this guess and not finding the -)- which it guesses should be there, the compiler issues the message 'EXPECT RIGHT PARENTHESIS'. Of course, a person looking at this line would see that putting in a right parenthesis is not a good way to correct the line, since it would still leave the rest of the line, namely '12-9' unexplained. With this clue a person would easily make the more illuminating guess that a comma was missing, and could then issue a more intelligent message like -MISSING COMMA-. However the compiler is much more myopic, easily confused, and the guesses which it makes when it encounters an error must therefore be taken very skeptically. About all that can be deduced from the appearance of an error message is that the line which it follows probably contains an error. This line should then be examined very carefully to see if you can spot the error. If in doubt as to what rules of SETL grammar apply, look up the relevant rules in the appropriate part of this book.

The diagnostic line following the line containing eight asterisks (namely the line starting with the word PARSING) which follows the line containing the word ERROR is actually of greater help than the first diagnostic line when you are trying to locate a minor grammatical error. In this line, the word PARSING is followed by the seven last 'tokens' (i.e., words, numbers, punctuation marks, or quoted strings) which precede the point at which the compiler was sure that an error had occurred. In our example, program line 3 is followed by the word PARSING, and then by the seven following 'tokens', which you will note occur in the program, just before the point of error:

```
45                                  (an integer)
```

```
)                                          (punctuation mark)
;                                          (punctuation mark)
PRINT                                      (a 'keyword')
(                                          (punctuation mark)
'The difference beween line and twelve is' (quoted string)
12                                         (an integer)
```

The compiler detected an error just between the last of these two tokens, where, as we know, a comma is missing.

It is normally not too hard to spot a grammatical error by looking carefully over the line to which an error message has been attached, and comparing it to the sequence of tokens following the word PARSING appearing in the second line of the error message, especially to the last few tokens of this sequence, which are likely to lie close to the actual point of error. However, this must be done with some caution, since after an error has occurred it may take a few lines of error-free program text for the resulting confusion (which affects the compiler) to dispel enough for additional error Messages to become accurate again. This phenomenon, a spurious error message issued in the wake of an initial error, is seen following lines 4 and 5 of our example program. In line 5, the perfectly correct END; has been flagged as an error since, coming as soon after the erroneous line 3 and 4 as it does, it is mistakenly taken as an illegal continuation of line 4 and not as an independent statement.

The manner in which the seven tokens following the word PARSING in the second line of an error message are underlined can also be helpful. Some of these tokens are underlined with hyphens, others with double bars, others not at all. The underlined symbols are those which are under active consideration at the moment when a grammatical error is detected. 'Reserved words', which cannot be used as variable names, and also punctuation marks, are underlined with double bars, other tokens with single bars. (This clue is valuable in cases in which you have accidentally used a reserved word as the name of a variable. See Appendix XXX for a list of all reserved words.)

Missing Quotation Marks

If you accidentally omit a quote mark (apostrophe) in your program, then whatever happens to follow the resulting unmatched quote mark will be taken as part of a quoted message (i.e., 'quoted string'). To prevent this rule from affecting the whole of your program, an arbitrary limit of 128 characters is established as the maximum permitted length of a quoted string; so recovery from this kind of error will normally take place a few lines later. When this kind of error occurs it will give a characteristically strange appearance to the list of tokens following the word PARSING in the very next error message; this should tip you off to the fact that the problem is a missing apostrophe

Comments preceded by dollar signs ('$', see Section X) are bypassed by the grammatical analysis process, and will never appear in the list of tokens following an error message. This can give such lists a different appearance from the program text to which they refer, especially if a comment many lines (or even pages) long has been bypassed.

Other features of the compilation history.

In your compilation listing, the lines that we have just been discussing are actually preceded by a largely blank page, containing just a few lines of information which looks approximately as follows:

```
    CIMS.SETL.PRS(81121)              THU  13 AUG 81  07:00:19   PAGE     1


       PARAMETERS FOR THIS COMPILATION:

       SOURCE FILE: I = DBC0:[NYUSETL.BERKOWITZ]TST.STL;2.
                    LISTING FILE: L = DBC0:[NYUSETL.BERKOWITZ]TST.LIS;1.
       POLISH STRING FILE: POL = TST.POL.
                    AUXILIARY STRING FILE: XPOL = TST.XPL.
       LIST DIRECTIVES: LIST = 1, AT = 1.
       PARSE ERROR LIMIT: PEL = 999. PARSE ERROR FILE: TERM = SYS$ERROR:.
       CHARACTER SET: CSET = EXT. MEASUREMENTS: MEAS = 0.
```

Don't pay too much attention to this material at first: it merely dates the listing and records various standard options which the compiler is using. You will only become concerned with these options (which are described more fully in Section 8.5.1) when you are working with long complex programs or want to secure one or another special effect.

Assuming that all goes well, the SEM phase will insert the following information into your output listing:

```
    CIMS.SETL.SEM(81121)              THU  13 AUG 81  07:00:22   PAGE     1


PARAMETERS FOR THIS COMPILATION:

POLISH STRING FILE: POL = TST.POL. AUXILIARY STRING FILE: XPOL = TST.XPL.
BINDER FILE: BIND = . IND. BIND FILE: IBIND = .
LITTLE Q1 FILE: Q1 = TST.LQ1. SETL Q1 FILE: SQ1 = .
SEMANTIC ERROR LIMIT: SEL = 999. SEMANTIC ERROR FILE: TERM = SYS$ERROR:.
GLOBAL OPTIMIZATION: OPT = 0. DIRECT ITERATION: DITER = 0.
USER DATA STRUCTURES: REPRS = 0.

       NO ERRORS WERE DETECTED.

       Q1 STATISTICS:
       SYMTAB(279,16383), VAL(242,16343), NAMES(746,16343).
       FORMTAB(52,2047), MTTAB(35,2047).
       CODETAB(23,8191), ARGTAB(33,16383), BLOCKTAB(3,1023).

       NORMAL TERMINATION.
```

This will be followed one page later by similar output produced by the COD phase, namely

```
    CIMS.SETL.COD(81099)          THU  13 AUG 81  07:00:30   PAGE     1

PARAMETERS FOR THIS COMPILATION:

LITTLE Q1 FILE: Q1 = TST.LQ1. SETL Q1 FILE: SQ1 = .
Q2 FILE: Q2 = TST.COD. SAVE INTERM FILES: SIF = 0.
CODEGEN ERROR LIMIT: CEL = 999. CODEGEN ERROR FILE: TERM = SYS$ERROR.
GLOBAL OPTIMIZATION: OPT = 0. BACKTRACKING: BACK = 0.
RUN-TIME ERROR MODE: REM = 2. ASSEMBLY CODE: ASM = 0.
CONSTANTS AREA SIZE: CA = 65535. SYMBOL TABLE SIZE: ST = 8191.
INITIAL HEAP SIZE: H = 600000.


       NO ERRORS WERE DETECTED.

Q2 STATISTICS:
MIN SYMTAB SIZE = 186. MIN CONSTANTS AREA = 47. MIN DYNAMIC HEAP = 483.
Q2 CODE SIZE = 38. INITIAL HEAP SIZE = 66018. MIN HEAP SIZE = 1088.
EXEC STATEMENTS = 4. Q2 INSTRUCTIONS = 19.
Q2 FORMAT DATE = 81099.

       NORMAL TERMINATION.
```

As for the PRS phase standard output, all this material merely records various standard options which are being used for compilation. Since both the SEM and (much more rarely) the COD phase of the SETL compiler can detect a few subtle errors in your code which the PRS phase may he missed, you will want at least to glance quickly at this output, to determine whether it ends with the line -- NORMAL TERMINATION -- signifying the absence of error. If not, the presence of errors is indicated. For an account of the errors which might be detected during the SEM and COD phas es, see Section XXX. Note however that errors in an earlier phase can cause mistaken error messages to be emitted by a later compiler phase. Thus, unless you have become expert in the use of the SETL system, you will only want to pay attention to error messages generated by the first compilation phase which detects any errors at all.

Note also that the output produced by your program follows <u>immediately</u> after the last line of standard material put out by the COD phase. Thus, especially if your program has produced only a few short lines of output, it is very easy to lose sight of your program's actual output, which may be concealed from your eye by the larger mass of standard material which precedes it. In this case, you may be confused into thinking that no output has been produced. Grow accustumed to looking for output quite carefully. The following shows the actual appearance of output from our sample program, in its physical relationship to the standard material produced by the compiler's COD phase.

```
    CIMS.SETL.COD(81099)          THU  13 AUG 81  07:00:30   PAGE     1

PARAMETERS FOR THIS COMPILATION:
```

```
LITTLE Q1 FILE: Q1 = TST.LQ1. SETL Q1 FILE: SQ1 = .
Q2 FILE: Q2 = TST.COD. SAVE INTERM FILES: SIF = 0.
CODEGEN ERROR LIMIT: CEL = 999. CODEGEN ERROR FILE: TERM = SYS$ERROR.
GLOBAL OPTIMIZATION: OPT = 0. BACKTRACKING: BACK = 0.
RUN-TIME ERROR MODE: REM = 2. ASSEMBLY CODE: ASM = 0.
CONSTANTS AREA SIZE: CA = 65535. SYMBOL TABLE SIZE: ST = 8191.
INITIAL HEAP SIZE: H = 600000.


       NO ERRORS WERE DETECTED.

Q2 STATISTICS:
MIN SYMTAB SIZE = 186. MIN CONSTANTS AREA = 47. MIN DYNAMIC HEAP = 483.
Q2 CODE SIZE = 38. INITIAL HEAP SIZE = 66018. MIN HEAP SIZE = 1088.
EXEC STATEMENTS = 4. Q2 INSTRUCTIONS = 19.
Q2 FORMAT DATE = 81099.

       NORMAL TERMINATION.
 99
 The difference of twelve and nine is: 3
 3025
```

Review of principal actions which occur when a job is run

The following summary lists all the principal system actions performed on your behalf between first submission of a program and the moment at which output produced by your program appears. Normally all this will proceed smoothly and require little attention on your part. However, trouble can occasionally develop, and then you will need to have at least some idea of all that is going on, if only in order to know whether the problems that have developed trace back to something wrong with your program or to difficulties elsewhere in the system:

1. User identity verified

2. Command lines analyzed and verified.

3. Operating systems programs (temporarily) pass control of computer to PRS phase of SETL compiler program, which reads, analyzes, and validates the SETL program which you have supplied.

4. PRS phase completes, producing listing as specified by initiating command including error diagnostics if any errors detected. Run may end if errors have been detected. Otherwise a data file representing the half-digested version of your program is saved for use by the next (SEM) compiler phase.

5. Operating system programs (temporarily) pass control of computer to SEM phase of SETL compiler, which continues analysis and translation of the SETL program which you have supplied.

6. Second (SEM) phase of SETL compiler is moved by operating system programs to the computer's central memory, and scheduled for execution.

7. The operating system programs (temporarily) pass control of computer to COD phase of SETL compiler, which completes the translation of the SETL program which you have supplied.

8. The SEM phase completes, adding to the output listing, and returning control to operating system programs. Additional error diagnostics may be tramnsmitted to the output listing. Otherwise a data file representing the partially translated version of your program is saved for use by the next (COD) phase of SETL compiler.

9. The third (COD) phase of the SETL compiler is moved by operating programs to the computer's central memory, and scheduled for execution.

10. The operating system programs (temporarily) pass control of computer to the COD phase of the SETL compiler, which completes the translation of the SETL program which you have supplied.

11. The COD phase completes, adding final compilation messages to output listing. Control is returned to the operating system programs, and a data file representing the internal, translated version of your program is saved for use by the SETL run-time system.

12. The SETL run-time system program is moved into central memory of the computer by operating system programs, and is scheduled for execution.

13. The operating system programs (temporarily) pass control of computer to the SETL run-time system, which follows the instructions found in the translated version of your program, producing output, and eventually either terminating, aborting if an illegal situation is found, or being forceably terminated by the operating system if it runs for long or produces too much output.

14. If your program is being run interactively from a terminal, the terminal will return to 'command mode' to await your next general instruction. If the program is being run on a 'batch' system, an additional 'dayfile' summary of system actions will be transmitted to the end of your output file which will then be released for printing. Later it will be printed and delivered to your standard output pick-up point.

$

CHAPTER 2

DATA AND EXPRESSIONS

This chapter has two parts. Sections 1 and 2 deal with the various kinds of data which the SETL language allows and is able to manipulate. The remainder, Sections 3 through X, describes the various kinds of expressions provided by SETL, using which new data objects can be formed. SETL provides data objects and expressions which are significantly richer than the objects provided in most other programming languages, so this chapter will be a bit longer than most others.

Chapter Table Of Contents

2.1. The main classes of data objects.

Like certain other programming languages, SETL allows one to manipulate two main kinds of data items, namely simple data items and composite data items. Four of the simple kinds of data items, namely

integers
floating point numbers
character strings
boolean values

are very much like those provided in most other programming languages. A fifth kind of data item, called 'atoms', will be a bit less familiar, but are still relatively easy to use. One very special quantity, namely the undefined value (called -OM-) is used frequently in SETL programs, and its somewhat nonstandard properties will become fully familiar as we go along. In addition to these simple data items, SETL provides exactly two kinds of composite objects, namely

sets

and

tuples

It is the fact that it allows sets to be used freely that gives SETL its name 'SET-L'.

Sets of one particular kind, namely sets of ordered pairs, play particularly important roles and therefore are sometimes referred to by a special term, namely

maps

These are all the classes of data values which the SETL language allows.

2.1.1 Integer, real, and boolean constants

To use objects of any of these kinds in a program we occasionally need to be able to write them out directly. For example, to give a variable x the value 3.14159 we may want to write

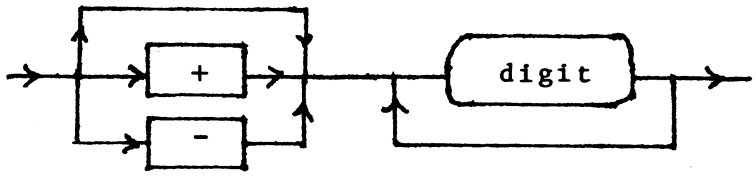
```
x := 3.14159;
```

A value written into a program in this way is called a constant, a constant denotation or (by some authors) a literal. The rules for the various forms of constants allowed in SETL are as follows:

(a) integers: Integers are written in the standard way, as sequences of decimal digits possibly preceded by a + or - sign. Examples are

```
 0
 1066
-50
+35
 001616232358
```

The way in which an integer denotation can be constructed can be summarized by means of a diagram, or graph, which looks as follows :



The diagram consists of rounded boxes, square boxes, and paths with arrows connecting these boxes. Each diagram has an edge that leads into it, and and edge that exits from it. A path through the diagram that follows the edges in the indicated directions is a valid instance of a language construct. The two kinds of boxes have the following meaning :

(i) A rounded box denotes a symbol of SETL, which must appear as is when used. For example, the + and - signs, the parenthesis, keywords such as IF, LOOP, EXISTS, and so on.

(ii) Square boxes correspond to other language constructs for which a separate diagram is provided. For example, the construct -digit- is described fully by a diagram that lists the 10 digits as valid instances of this construct. A full list of diagrams for SETL is provided in Appendix A. To test your understanding of these, verify that the diagram presented above allows you to write -12345678 as a SETL integer, but forbids ->12345678.

(b) floating point numbers: Floating point numbers are written in one of the notations that have become standard, namely either in decimal form or in exponent form. A real number in decimal form is a sequence of decimal digits, followed by a decimal point, followed by a second sequence of decimal digits, and possibly preceded by a + or - sign. The initial but not the final sequence of digits can be omitted. Examples are

```
 0.0
  .3156    (but note that 3. is illegal)
 1066.6
-50.50
+35.50
 3.1415928
```

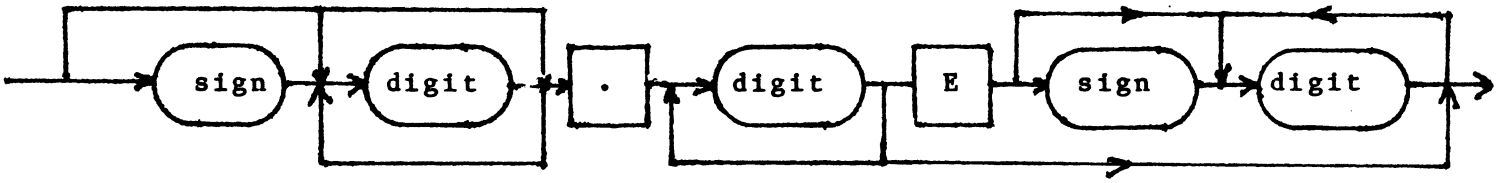
A real number in exponent form is a real number in decimal form, immediately followed by the letter E, and then by an integer (the exponent). Examples are

```
1.0E100
31415.9E-4
6.0E+23
```

This last form for real constants corresponds to the ordinary 'scientific' notation for decimals, e.g. these three examples would be written in ordinary scientific notation as

$1.0 * 10^{100}$, $31415.9 * 10^{-4}$ and $6.0 * 10^{23}$.

The previous description of floating point is summarized by the following diagram:



This diagram makes it clear that any valid floating point constant must have one digit or more after the decimal point, but may have none before it.

(c) string: A string is an ordered sequence of zero or more characters. To write a string as a constant we enclose it within (single) quotes (i.e., apostrophes) as in the following examples:

```
'Brother, can you spare a dime?'
'*!!-;*!!'
''
```

This last example shows the null string, i.e., the (unique) string consisting of zero characters. Note that blanks appearing within a string are significant, i.e., are treated in the same way as any other character. Thus, although the number of characters in 'Hello' is 5, the number of characters in 'Hello ' or ' Hello' is 6, and the number of characters in ' Hello ' is 7.

If the quote mark (i.e., apostrophe) itself is to appear within a string s it must be written doubled, to indicate that it is part of s and not the end of s. Thus, to write the string - Mary's mom - as a constant, we would write

```
'Mary''s mom'
```

Note that the doubled apostrophe after the letter -y- serves to denote a single apostrophe in the actual string constant.

Any of the characters available in the machine which you are using can be used in a string constant, although SETL programs which are to be run on a variety of different computers should restrict themselves to the

characters available on all computers to avoid character-set translation problems.

Sometimes one will need to write a long string constant, so long that it must cross a line boundary. This can be done by ending the first part of the string with a quote (i.e. apostrophe) and then continuing immediately on the next line, with a second quote character to continue the string. This "line break" sequence is called a string continuation and is not included in the actual string value of the multiline string constant. This means, for example, that we can write the string assignment statement

```
x :='Brother, can you spare a dime?'
```

on two lines as

```
x := 'Brother, can you spa'
        're a dime?'
```

when there appears to be any reason to do so.

(d) Boolean values: There are two Boolean values, truth and falsity, in SETL. These are written as TRUE and FALSE respectively. These values are typically produced as the results of tests, e.g. the value of the expression (3 > 1) is TRUE and the value of (1 < 3) is FALSE.

(e) atoms: atoms are generated names, or tags, that can be used to label objects in a SETL program. Atoms are different enough from other data types in their functions and use, that we will postpone their discussion until Sec. 2.8.

Let us now discuss the rules for writing constant composite objects, namely sets and tuples.

2.1.2 Constant sets.

Sets in SETL are finite collections of arbitrary values. To write a set constant, we simply list the members of the set, with commas between successive members, within the set brackets '{' and '}'. Three examples are:

```
{1,2,3,4}
{'Tom', 'Dick', 'Harry'}
{TRUE,FALSE}
```

The first of these is the set of all integers between 1 and 4; the second is a set of three strings, namely, 'Tom, 'Dick', and 'Harry'; the third is the set consisting of the two possible boolean values TRUE and FALSE.

The 'null' or 'empty' set, i.e. the (unique) set having no members at all, is a legal SETL value. It is written as follows:

```
{ }
```

Any legal SETL value (with the sole exception of the undefined value OM) can

be a member of a set. Examples illustrating this are

```
{1,TRUE,'Tom'}
{1,TRUE,'Tom', {3}}
```

The first of these two examples is a perfectly legitimate set whose three members are the integer 1, the Boolean value TRUE, and the string 'Tom'. The second has four elements, the integer 1, Boolean value TRUE, string 'Tom', and the set {3} , i.e., the 'singleton' set whose sole member is the integer 3 . This shows that sets need not be homogeneous, i.e. are not restricted to have members all of the same kind, and that sets can be members of other sets. Note also that the integer 3 is not a member of the set {1,TRUE,'Tom', {3 }} , but that the set {3}, which is quite a different thing, is. A more complex example illustrating this same fact is

```
(*)      {1, {2}, {{3}}, {}, {5,6}}
```

This is a set of five members, namely: the integer 1, the set {2} whose sole member is the integer 2; the set {{3}} , whose sole member is the set {3} ; the null set {} , and the set {5,6} consisting of the integers 5 and 6. Note that in this example the integer 3 is neither a member nor a member of a member of set (*); rather, it is a member of a member of a member of (*).

As ordinarily in mathematics, set values never actually contain duplicate members, and the members of a set have no implied order. Thus the sets {1,1} and {1} , both of which are legal, designate exactly the same set, namely the set whose sole element is the integer 1. Similarly, {1,2} and {2,1} designate the same set, namely the set whose members are the integers 1 and 2. For a more complex example, note that

```
{1, 2, {3,4}}
```

and

```
{{4,3},2,1}
```

designate the same set, namely the set whose three elements are the integers 1 and 2 and the set {3,4} (but {1,2,3,4} , which is a set of four elements, namely the integers 1 through 4, is different).

Since the elements of a set are not considered to have any particular order within the set, it is incorrect to speak of the first, second, or last element of a set. That is, it is incorrect to speak of the string 'Tom' as the first element of the set

```
{'Tom', 'Dick', 'Harry'}
```

or to speak of the string 'Harry' as its last element, since this same set can as well be written as

```
{'Harry', 'Tom', 'Dick'}
```

or

```
{'Dick', 'Tom', 'Harry'}
```

In working with sets, one must always remember that their elements have no particular order, and that duplicates are eliminated.

2.1.2.1 Sets of successive integers.

Sets whose elements are successive integers, such as :

{1,2,3,4,5,6,7}, {-3,-2,-1,0,1,2,3}

arise often enough that a special notation is provided for them. To describe the set of all integers lying in the range M to N inclusive, where M and N are integers, we write :

{M..N}

The two dots (not three, and not commas!) stand for all the integers M+1, M+2, and so on up to N-1. Sets of integers of the form :

{1,3,5,7,9} or {10,5,0,-5,-10,-15}

that is to say, sets that represent an arithmetic progression, are also useful enough to be given their own notation in SETL: We represent such sets by giving the first, second, and last element of the progression, as follows:

{1,3..9} {10,5..-15}

note again the use of two dots to indicate middle part of the sequence. These notations will be used frequently in what follows.

When sets are printed, their elements can appear in any arbitrary order. For example,

```
print({1..10}) ;
```

might be expected to produce {1,2,3,4,5,6,7,8,9,10} . However, if you try it out, you will see the following appear :

{4,5,6,7,1,2,3,9,10}

(or perhaps some similar permutation of the integers from 1 to 10). This emphasizes the fact that the elements of a set have no particular order; the set {1..10} contains the integers in the range 1..10, but in the set these integers have no particular order.

2.1.3 Tuples.

In contrast to sets, tuples (sometimes also called vectors) in SETL are finite ordered sequences of arbitrary elements. To write a tuple constant, we simply list its successive components, in order, within the tuple brackets '[' and ']'. Components in such a list are separated by commas. Three examples are

```
[1,2,3,4]
['Tom', 'Dick', 'Harry']
```

[TRUE, FALSE]

The successive components of a tuple, as distinct from the elements of a set, do have a definite order within the tuple. Thus a tuple is a quite different kind of object from a set, even though the components of the tuple may all be elements of the set, and vice versa. As an example of these rules, note that

[1,2,3,4] and {1,2,3,4}

are regarded in SETL as entirely different objects, and, indeed, as objects of entirely different types; the first is a tuple, the second is a set. Note also that [1,2,3,4] and [2,1,3,4] are different objects, since the components of a tuple are considered to have a specific order and two tuples are only equal if they have the same components in the same order; however, the sets {1,2,3,4} and {2,1,3,4} are the same, since a set, as distinct from a tuple, is defined by the collection of its elements, not by their order.

Tuples, like sets, need not be homogeneous, i.e. the components of a tuple need not all be of the same type. Tuples can have sets as their components and sets can have tuples as their members. Indeed, sets and tuples can be nested within each other to arbitrary depth as members and components, permitting construction of a great variety of data objects. Examples are

```
(1)     [1, 'Tom',   {'Dick'}, ['Harry']]
(2)     { 1, 'Tom', ['Dick'], {'Harry'}}
(3)     [1, {'Tom', ['Dick', 'Harry' ]} ]
```

The first of these constants represents a tuple of four components, which, in order, are the integer 1, the string 'Tom', the singleton set {'Dick'} , and the one-component tuple ['Harry']. The second represents a set of four elements, which (in no particular order) are the integer 1, string 'Tom', the one component tuple ['Dick'], and the singleton set {'Harry'} . The third represents a tuple of just two components, namely the integer 1, followed by the two-element set {'Tom', ['Dick', 'Harry']}. We can therefore assert that the string 'Harry' is the first (and only) component of the fourth component of the tuple (1); that 'Harry'is also a member of a member of the four-element set (2); and finally that 'Harry' is a member of the second component of a member of the second component of the tuple (3).

Another example of a perfectly legal though highly nested SETL construction is

{{{{}}}}

this designates a set (let's call it s), and the empty set is the only member of the only member of the only member of s. Such constructs are used occasionally (though rarely) in real SETL programs.

Repetition of tuple components, as distinct from repetition of set elements is logically possible and changes the tuple value. For example the three tuples

['Tom'] , ['Tom','Tom'] , and ['Tom', 'Tom', 'Tom']

are all distinct; the first has just one component and is of length 1; the second is of length 2; and the third is of length three, and has three components: its first, second, and third components are all defined, and each of them is the string 'Tom'. In contrast, the constants

{'Tom'}, {'Tom','Tom'}, and {'Tom','Tom','Tom'}

designate the same set, which has just one element, namely the string 'Tom'. Since tuples, as distinct from sets, are considered to have a definite order, It does make sense to refer to the 'first', 'second', ..., 'last' component of a tuple. For example, the first component of

['Tom', 'Dick', 'Tom', 'Tom']

is the string 'Tom'; its last (also fourth) component is also 'Tom'; its second component is 'Dick'.

There is a (unique) 'null' or 'empty' tuple, which is written as

[]

This plays much the same role for tuples that the important null set, i.e. {}, plays for sets.

2.1.3.1 Tuples of sequences of integers.

Tuples whose components constitute an arithmetic progression can be written in a special SETL notation similar to that used for sets of integers. The tuple construct:

[M..N]

where M and N are integers, describes the tuple whose components are the integers M, M+1, M+2 and so forth, up to N. If N is less than M, this construct is equivalent to the empty tuple.

Similarly, an arithmetic progression of the form

M, M+k, M+2*k, .. N

where k is some integer (positive or negative), can be described by writing its first, second and last component; specifically, the tuple whose components constitute such a sequence can be written as:

[N, N1 .. M]

where M1, the second term in the sequence, has the value (M+k). For example, the construct [3,6..600] represents a tuple whose components are the first 200 positive multiples of 3, in increasing order. This construct, and the related set construct {N, N1..M}, are simple instances of a general numeric iterator construct, which will be discussed in detail in Sec.3.x.y.

2.1.4 Maps.

In SETL a map is simply a set all of whose elements are pairs, i.e. are tuples of length 2. Some properties of maps can be deduced from their structure, I.e. from the fact that all their components are pairs. But maps are important enough to have a number of operations that apply solely to them. We will see that maps are one of the most expressive programming features of SETL, and that the proper use of maps is a hallmark of good SETL style. Maps allow us to associate elements of various collections of objects: countries with their capitals, numbers with their cubes, people with their dates of birth, courses with their sets of students, and so forth. Suppose for example, that the children in a family, listed in increasing order of age, are

Sue, Tom, Mary, Alphonse.

Suppose that we want to associate each child x in this family with the number of younger sisters that x has. For this purpose, we could use the following map:

(1) { ['Sue',0], ['Tom',1], ['Mary',1], ['Alphonse',2]} .

Similarly, the map

(2) {['Sue',0], ['Tom',0], ['Mary',1], ['Alphonse',1]}

associates each child x with the number of younger brothers that x has. The map

```
(3)   {['Sue',  {'Mary'}],['Tom',  {'Sue','Mary'}],
                ['Mary',  {'Sue'}], ['Alphonse',  {'Sue','Mary'}]}
```

associates each child x with the set of sisters of x. Note therefore that maps can be used to associate values of any type with other values of any type.

Another interesting map is

```
(4)   {['Sue','Mary'], ['Tom','Sue'], ['Tom','Mary']},
      {['Mary','Sue'], ['Alphonse','Sue'], ['Alphonse','Mary']}.
```

This contains a separate pair associating each child x with each of the sisters of x (rather than one pair associating x with the set of all the sisters of x (as in (3); (3) and (4) are different, but closely related and record much the same information). Since several different pairs in (4) (e.g. ['Tom','Sue'] and ['Tom','Mary']) have the same first component, (4) is called a multivalued map. Maps for which this does not happen, i.e. in which no two distinct pairs share the same first component, are called single-valued maps.

Given a map M, we can form the set D of all first components of pairs in M. This is called the domain of M, and is written

DOMAIN M

We can also form the set R of all second components of pairs in M, which is called the range of M and is written

RANGE M

The following table shows the domain and range of the maps appearing in examples just presented.

map number	domain M	range M
(1)	{'Sue','Tom','Mary','Alphonse'}	{0,1,2}
(2)	{'Sue','Tom','Mary','Alphonse'}	{0,1}
(3)	{'Sue','Tom','Mary','Alphonse'}	{'Sue'}, {'Mary'}, {'Sue','Mary'}}
(4)	{'Sue','Tom','Mary','Alphonse'}	{'Sue','Mary'}

Maps and the map-related operations of SETL, which will be presented in Section X below, are the most characteristic and important features of the language.

Be sure you understand the rules and distinctions concerning sets and tuples, duplicates, ordering, nesting, and maps presented in the preceding pages. Review this material if necessary, and work the exercises of Section 2.2. This material must be mastered before proceeding, since it will be used constantly in all later chapters.

2.1.5 The size of composite objects: the # operator.

One of the most important characteristics of a composite object is the number of components which it has. SETL provides provides a single operator to determine the size of sets, tuples, maps and strings : the '#' operator. The '#' operator is called indifferently length, size, or cardinality.

When applied to a string it yields its length, i.e. the number of characters is contains; when applied to a tuple, it yields the length of the tuple, i.e. the largest position in the tuple that is occupied by a component whose value is not OM ; and when applied to a set it yields its cardinality, i.e. the number of its elements. For a map, it yields the number of pairs in it . Thus

#'Tom'	is 3, since 'Tom' has 3 characters
#'Tom is hot'	is 10, since 'Tom is hot' has ten characters (including 2 blanks)
#['Tom','Dick','Harry']	is 3, since this tuple has 3 components
#['Tom','Tom','Tom']	is 3, since this tuple also has 3 components
# {'Tom','Dick','Harry'}	is 3, since this set has 3 elements
# {'Tom','Tom','Tom'}	is 1, since this set has 'Tom' as its only member
#{}	is 0, since the null set has no members

`#[]`	is 0, since the null tuple has no components
`#''`	is 0, since the null string contains no characters
`# {[4,2], [4,-2] [0,0]}`	is 3, becuase this set (or map) contains three elements (pairs).

2.2 Exercises

1. Which of the following objects are the same, and which are different?

(1.a)	'The'	and	'The '
(1.b)	'The man'	and	'Theman'
(1.c)	['The','man']	and	['man','The']
(1.d)	{'The','man'}	and	{'Man','The'}
(1.e)	{'The man'}	and	{'man The'}
(1.f)	{ 'The','The','man'}	and	{'The','man'}
(1.g)	['The','The','man']	and	['The','man']
(1.h)	['The','man']	and	{'The','man'}
(1.i)	['The','man']	and	{'The, man'}

2. Write the size #x of the following strings, sets, and tuples. For each set and tuple, also write the list of all its integer elements or components and the size of each of its set, tuple, or string elements or components.

(2.a) {1,2,2,'Tom' }
(2.b) [1,2,2,'tom']
(2.c) {1,{2,2} ,'Tom'}
(2.d) { 1, 1 , { } , {}}
(2.e) [{} ,[[]]]
(2.s.f) 'abracadabra'
(2.g) 'abra cadabra'
(2.h) 'abra, cadabra'
(2.i) {1,'abra','cadabra'}
(2.j) {1,'abra''cadabra'}
(2.k) {1,'abra,cadabra'}
(2.l) {1,'abra','cadabra' }
(2.m) {1,'abra'',''cadabra'}
(2.n) {[],'', {} ,'[]',' {[]} ', '{}'}

3. Write the size of the first, second, and last component of each of the following tuples:

(3.a) ['Tom','Dick','Harry']
(3.b) ['Tom','Dick','Harry','Tom']
(3.c) ['Tom',['Tom'],'[Tom]','[]','','''']

4. Indicate whether Tom is a member, component, member of component, component of member, component of component, etc. of each of the following sets or tuples:

(4.a) [1,'Tom']
(4.b) { ['Tom',3],['Dick',4],['Harry',5]}
(4.c) { {'Tom','Dick','Harry'}}
(4.d) [[[['Tom'],'Tom'],'Dick', 'Tom', 'Harry']]
(4.e) ['Tom','Dick'],'Tom','Harry']

5. Write a map which indicates the age of each of your brothers and sisters by associating their age with their first name.

Write the range and domain of this map.

6. Write a map which associates each component of the tuple ['Tom','Dick','Harry'] with the square of the component length. Write the range and domain of this map.

7. How many maps are there whose domain is {'Tom','Dick'} and whose range is {'Sue','Mary'}? How many of these maps are single-valued?

8. A map M associates the age of each child in a family with the name of the child. The domain of M is {7,9,13} and the range is {'Sue','Mary','Tom','Dick'} . What is interesting about this family?

9. Consider the following map M:

```
 {['Smith',  {['Sue',11],['Jim',13]}  ],
['Jones',  {['Albert',1],['Anna',3],['Ron',9]} ],
['Skallagrim',  {['Thorolf',7],['Egil',5],['Asgerd',4]} ] }
```

What information might this map represent? What is its domain? What is its range?

10. Let S be the set {'Tom',{'Dick,['Harry','Arthur',{'Tom'}]}} 'Dick' is a member of S. Match each name in the following list with the manner in which it appears in S:

(a)Tom	(i)component of member
(b)Harry	(ii)member of component of member
(c)Arthur	(iii)member

11. Consider the map M as a set. What are all the members of this set? Which of the components of the members of M are sets, and what are the members of these members? What are all the components of the members of all the components of the members of M which are sets? What are all the lengths of all the components of the members of M which are not sets?

12. Write a map which associates each of the Pacific coast states with the name of its state capital.

13. For how many integers between 1 and 100 is I=5*(I DIV 5) true? For exactly which integers is this true? For how many integers between 1 and 100 is I=(5*I) DIV 5 true?

2.3 Expressions and Statements

The use of expressions like those of algebra are one of the main features of many programming languages, including SETL. Expressions denote values, which can be printed, saved as the values of variables, etc. The following are typical (though simple) expressions:

```
3+5*(7-11)

17.0/31.3131 + 19.9

x + y

x1+x2+x3+y1*y2*y3
```

As these examples show, an expression can involve both constants and variables (also called identifiers). Values are given to variables by assignments, of which the following, which assigns the value 3 to the variable zz1, is typical:

```
zz1 := 3;
```

Note that an assignment is written using the := (colon-equal) sign, sometimes called the assignment operator. The assignment is the first type of statement that we will use. Statements are the basic building blocks out of which programs are constructed. In this chapter we will only use two types of statements : the assignment statement, and the print statement, whose purpose is to display (on the screen, or on an output listing) the result of a computation. The print statement has the format :

```
PRINT(expression1,expression2...) ;
```

that is to say, it consists of the keyword PRINT, followed by a list of expressions, enclosed between parentheses, and separated by commas. Any number of expressions can appear in a print statement. A print statement that does not include a list of expressions will simply produce a blank line.

A variable appearing in an expression always stands for its current value. Thus, if we write the commands

```
zz1 := 3;
zz2 := 17;
print(zz1) ;
print ;
print(zz1+zz2);
```

The current value of the variables zz1 and zz2 at the moment that the -print- instruction is executed will be 3 and 17 respectively, so that the output of this program fragment will be :

```
3
20
```

(Note the blank line separating the two printed values).

Suppose next that we write the commands

```
zz1 := 3;
zz2 := 17;
print(zz1);
zz1 := 4;
print(zz1 + zz2);
print(zz1)
```

This will produce the output

```
3

21
4
```

because the value of the variable zz1 has been changed by the assignment statement 'zz1 := 4' after the first print statement but before the second 'print' statement, and because (we say it again) a variable apparing in an expression always stands for its current value, i.e. the last previous value given to the variable by any assignment (or assignment-like) statement. Do not go on before you understand this point. To test yourself, see if you can tell what output the following sequence of command will produce:

```
x := 1;
print(x);

s := 2;
print(x);

y := 3;
print(x + y);

x := 0;
print(x + y);

y := 0;
print(x + y);

x := 1;

print(x + x);
y := 1;

print(x + x);
print(x + y);
```

Expressions can be compounded, that is, an expression e1 can be substituted for any variable appearing in another expression e2, thereby generating a more complicated but still legal expression. For example, by substituting x+y for z in 2*z, one generates the expresion 2*(x+y). Then, by substituting 3*a*b for y in the result, one generates the expression

2*(x+3*a*b).

As in algebra, the order in which a compound expression containing many operators is evaluated is determined by the 'precedences' of the operators involved, as modified by the rule that subexpresions enclosed within parentheses must always be evaluated before any operation is applied to them. Multiplication and division are given higher precedence than addition and subtractions, and are therefore performed before the latter. For example, 1+2*3 has the value 7 rather than 9, because the multiplication 2*3 is performed before the addition; but (1+2)*3 has the value 9 since the parentheses force the addition to be performed first.

Both binary operators like the '+' in x+y, and unary operators like the '-' in x+(-y) can appear in expressions. As these examples indicate, some operator signs like '-' can designate both binary and unary operators: unary if they are preceded by a left parenthesis or by another operator, binary otherwise. On the other hand, some operator signs are only used to designate binary operators, while others are only used to designate unary operators. All the (binary and unary) SETL operators will be described in this Chapter and in Chapter V, and are summarized for ready reference in Section XXX. Section 2.11 contains a table giving the precedences of all operators.

2.3.1. Variable identifiers

Almost all programming languages make it possible to perform calculations and then save their results for re-use later. This is done by assigning the results of calculations to a variable identifier (sometimes abbreviated simply as variable, or as identifier). An example is

```
x := 1 + 2 + 3 + 4 + 5;
```

which saves the result of the expression 1 + 2 + 3 + 4 + 5 appearing to the right of the assignment operator :=, making the result the value of the variable identifier x appearing to the left of this assignment operator. Since the value in question is 15, the command

```
PRINT(x);
```

would then print the current value of the variable x, namely 15.

Identifiers are composed of the letters, digits, and the underscore character '_'. The first character of an identifier must be a letter. The following are examples of valid identifiers:

```
x
x23
big1
End_of_Input_flag
set_OF_garbage_symbols
z123456789
eta_
```

On the other hand, the following are not valid identifiers:

```
big 1
x.23
23x
```

because the first two contain characters other than letters, digits, and underscores (blank in the first case, period in the second), while the third begins with a digit rather than a letter.

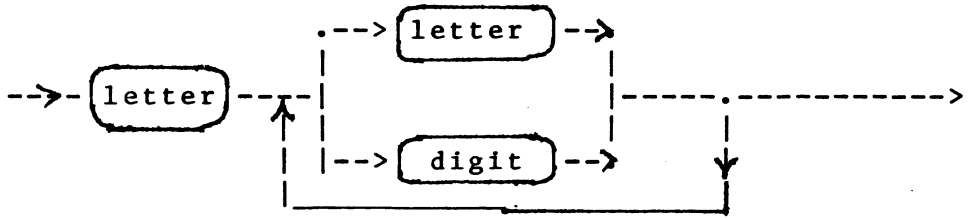
Identifiers can be of any length, but cannot be split beween two lines.

Except within quoted string constants, capitalization is ignored by the SETL compiler. Thus all the identifiers

```
Big_set
big_set
BIG_SET
big_SET
BiG_sEt
```

are considered to be identical.

The following diagram describes the structure of valid identifiers :



The proper choice of identifiers can make an important contribution to the clarity and professionalism of your programs. If you choose identifiers thoughtfully, your program will be easier for others to read and understand, and, equally important, will be easier for you to understand. Careless errors are also less likely to occur, since the inner 'rhythm' of a well-chosen set of identifiers will make errors easier to detect when your program is written, typed, and proofread. Here are some useful guidelines for the choice of identifiers:

(a) Choose 'mnemonic' identifiers, i.e. identifiers which explain the meaning of the quantities which they represent. E.g., an identifier which represents some sort of upper limit value in a program should be called upper_limit or uplim rather than simply u or L.

(b) Avoid ambiguity in the choice of identifiers, and use standard spellings. It is certainly bad practice to have two different identifiers called, e.g., STACK and STAK. It is also bad practice to use variant spellings like STAK, since without noticing it you may slip back to the standard spelling. Use the standard spelling STACK instead. (Note in connection with (a) and (b) that some of the SETL dump facilities, which when switched on (see section X) print out information useful for pinpointing program errors, truncate identifier names to eight characters. It is therefore a good idea to ensure that variable names used in contiguous contexts can be identified using their first eight characters only, i.e. use names like TABLE_1_IDENTIFIER and TABLE_2_IDENTIFIER rather than TABLE_IDENTIFIER_1 AND TABLE_IDENTIFIER_2, which could not be told apart in an error dump.)

2.3.2 Integer operators: +,-,*,**,DIV,MOD,=,/=, >, <>, >=, <>=, MAX, MIN, ABS, EVEN, ODD, FLOAT, RANDOM.

We begin our systematic description of the operators SETL by discussing those operators that take arguments of integer type. Some of these operators yield a value of the same type: for example the familiar arithmetic operators of addition, subtraction, multiplication and division. Another group of integer operators yields a truth value : TRUE or FALSE. This is the case for the comparison operators (Greater than, equal to, etc.) These operators are often called predicates. Finally, a conversion operator, namely FLOAT, allows us to convert an integer into a floating point quantity. The binary integer operators provided by SETL are as follows:

i+j	computes the sum of i and j
i-j	computes the difference of i and j
i*j	computes the product of i and j
i**j	computes i to the jth power. An error results if j is negative or if i and j are both zero.
i DIV j	computes the integer (whole number) part of the quotient of i by j. The fractional part of the quotient is simply discarded. An error results if j = 0. See the examples given below for the way in which i DIV j works if one of i or j is negative.
i MOD j	computes the remainder left over when i is divided by j. An error results if j = 0 . The result is always positive.
i MAX j	yields the larger of i and j
i MIN j	yields the smaller of i and j.

Integer predicates

i = j	yields TRUE if i and j are the same, FALSE otherwise
i /= j	yields TRUE if i and j are different, FALSE otherwise
i > j	yields TRUE if i is bigger than j, FALSE otherwise
i < j	same as j > i
i >= j	yields TRUE if i is no smaller than j, FALSE otherwise
i <= j	same as j >= i

Examples of use of these operators are

```
print(1+1);                    yields  2
print(1-1, 1-10);              yields  0  -9
```

```
print(1*2,1*(-2),(-1)*2,(-1)*(-2)); yields  2  -2  -2  2
print(2**3,(-2)**3,2**0,(-2)**0);   yields    8  -8   1  1
print(1 DIV 3, 2 DIV 3, 3 DIV 3,
      4 DIV 3);                     yields    0   0   1   1

print(1 MOD 3, 2 MOD 3, 3 MOD 3,
      4 MOD 3);                     yields    1   2   0   1
print(7 DIV 3, (-7) DIV 3,
     7 DIV(-3), (-7) DIV(-3));      yields    2  -2  -2   2
print(7 MOD 3, (-7) MOD 3);         yields    1   2

print(1 MAX 2, (-1) MAX (-2));      yields    2  -1
print(1 MIN 2, (-1) MIN (-2));      yields    1  -2
print(1 = 1, 1 = 2);                yields    TRUE  FALSE
print(1 /= 1, 1 /= 2);              yields    FALSE TRUE

print(1  > 1, 1  > 2, 2 < 1);       yields    FALSE  FALSE  FALSE
print(1 > 1, 1 < 2, 2 < 1);         yields    FALSE  TRUE   FALSE
print(1  >= 1, 1  >= 2, 2  >= 1);   yields    TRUE  FALSE  TRUE
print(1 >= 1, 1 <= 2, 2 <= 1);      yields    TRUE  TRUE  FALSE
```

Concerning i DIV j and i MOD j, it is useful to note that for i (and j) positive we always have i = (i DIV j) *j+(i MOD j), but for i negative this is false, e.g.

```
        (-7) DIV 3        is -2,
but
        (-7) MOD 3        is  2.
```

Unary integer operators compute a result value from a single input i. Two of these operators are predicates, namely ODD and EVEN. The unary integer operators provided are as follows:

+i	has the same value as i
-i	computes the negative of i
ABS i	computes the absolute value of i
EVEN i	yields TRUE if i is even, FALSE if i is odd
ODD i	yields FALSE if i is even, TRUE if i is odd
FIX i	converts the floating-point (i.e. real) number i to the corresponding integer value. (See Section 5.1 for a discussion of real numbers).
FLOAT i	converts the integer i to the corresponding floating point (i.e. real) value. (See Section 5.1 for a discussion of floating-point numbers). If the conversion causes overflow, which is possible if i has a very large value, then an error results.

RANDOM i returns an integer selected at random from the range from zero to i, including both end points. For example, RANDOM 5 will give one of the six integers 0,1,2,3,4,5. Successive uses of this operator will in general give different randomly selected values.

Examples of these unary operators are:

```
print(+1, +(-100));                          yields   1 -100

print(-1, -(-100));                          yields  -1  100

print(ABS 1, ABS(-2));                       yields   1   2

print(EVEN 1, EVEN 2, EVEN (-1));            yields   FALSE  TRUE  FALSE

print(ODD 1, ODD 2, ODD (-1));               yields   TRUE  FALSE  TRUE

print(FLOAT 1, FLOAT (-1), FLOAT 2);         yields   1.0  -1.0  2.0

print(RANDOM 5, RANDOM 5, RANDOM 5);         yields   0   4   3 , or
                                             some other sequence of integers
                                             chosen independently and at random
                                             from the range 0 through 5
                                             inclusive.

print(RANDOM(-5),RANDOM(-5),RANDOM(-5));     yields  -2  0  -4  or
                                               some other sequence of integers
                                               chosen independently and at random
                                               from the range 0 through -5
                                               inclusive.
```

2.3.2.1 Exercises

Ex. 1 What output will be produced by the following code?

```
Program one ;
 x:=1;y:=2
 print(x+y);
 x:=3;
 print(x+y);
 y:=x+y;
 print(x+y);
END ;
```

Ex.2 What is the output producd by the following program ?

```
PROGRAM multiply_x_by_y;
x:=1;y:=2;
print(xy);
END;
```

Ex. 3 What output will the following code produce?

```
program thr3 ;
 number:=1; Number:=2; NUMBER:=3;
 print(number+Number+NUMBER);
 number:=number*NUMBER;
 print(number+Number+NUMBER);
END ;
```

Ex. 4 Which of the following are valid identifiers?

(4a) number_1 (4b) number 1 (4c) number.1

Ex. 5 What output will the following code produce?

```
PROGRAM five;
 number1:=1; NUMBER1:=2; Number_1:=3;
 print(number1+Number_1+Number1);
 number1:=Number1*Number-1;
 print(number_1+Number1+NUMBER1);
END PROGRAM;
```

Ex. 6 What output will the following code produce?

```
PROGRAM xs;

  x:=1; y:=2; z:=3; w:=4;
  print(x+y), z*(x+y), z*x+y, w+z*(x+y));
  w:=2;
  print(w+z*x+y,z*y/w, y**(x+y)*z);

END PROGRAM xs;
```

Ex. 7 Which of the following are valid expressions?

```
(7.a) x  (7.b)  x+y  (7.c)  (x+y)**w
(7.d)  (x+y)**w**w  (7.e)  a_1 DIV (x+y)**w**w
```

Ex. 8 Evaluate the following constant expressions:

```
(8.a) 2**2          (8.b) 2**2**3          (8.c) (2**2)**3
(8.d) 2**(2**3)     (3.e) 3 DIV 2          (8.f) 1 DIV 2
(8.g) (1+2) DIV 4   (8.h) (-11) MOD 5      (8.i) -11 MOD 5
(8.1) 2**2**3/=64   (8.m) 3-0 / 3          (8.n) 3-0<3
(8.o) (-35) MIN 1
```

Ex. 9 Simplify the following expresions:

```
(9.a)  +-+--x
(9.b)  ----x
(9.c)  x MAX y MIN y
(9.d)  x MAX (y MIN y)
(9.e)  x MAX x
```

Ex. 10 Evaluate the following constant expressions:

```
(10.a)  ABS -1 + ABS -2  (10.b)  ABS(-1 + ABS -2)
(10.c)  ABS (1 MIN -1)   (10.d)  ABS (1 MAX -1)
(10.e)  1 MIN 2 MIN 3    (10.f)  1 MAX 2 MAX 3
(10.g)  2 + 2 MAX 3 + 3  (10.h)  -2 -2 MAX -3 -3
```

Ex. 11 Re-express the following expressions in as simple a way as you can using the MAX, MIN, and ABS operators:

```
(11.a)  x MAX -x  (11.b) x MIN -x
(11.c)  (x MAX o) + (x MIN 0)
(11.d)  (x MAX 0) + (-x MAX 0)
```

2.3.3. String operators: S(i), S(i..j), S(i..), +, *, =, /=, >, <, >=, >=, #, ABS, CHAR, STR

Binary string operators compute a result value from two inputs, at least one of which is a string. Some of these operators take two strings as their arguments, while others take a string and a positive integer as their arguments. Some of these operators are predicates, and perform string comparisons analogous to the integer comparisons discussed above.

In what follows, s and ss are always strings, while i and j are integers.

The string operators are the following:

s(i)	computes the i-th character of the string s; the result is a one-character string. If i is negative, an error results; if i is greater than the length of s, then the value OM is returned.
s(i..j)	this 'string slice' operator computes and returns the substring of s which extends from its i-th through its j-th characters, inclusive. If i = j-1, a null string is returned. See Table 2.1 below for a description of the treatment of other marginal and exceptional cases for this operator. (Note that this operator actually has three, rather than two, arguments.)
s(i..)	this computes and returns the substring of s which extends from its i-th character through the end of s, inclusive. See Table 2.1 below for a description of the treatment of marginal cases of this operator.
s + ss	concatenates the two strings s and ss.
i * s	concatenates i successive copies of the string s. If i = 0, then i * s is he null string. If i < 0 then an error results.
s = s	yields TRUE if s and ss are identical, FALSE otherwise.
s /= ss	Yields TRUE is s and ss are distinct, FALSE otherwise.
s > ss	yields TRUE if s comes later than ss in standard alphabetical order, FALSE otherwise. (Note that this operation, as well as the other string comparisons s < ss, s >= ss, s <= s' are implementation dependent, as they depend on an assumed alphabetical order of characters ('collating order'). Of course, alphabetic characters will always have their standard order, but the relative order of punctuation marks, and also the way in which alphabetics compare to numerics, may differ from implementation to implementation.)
s < ss	yields TRUE if s comes earlier than ss in standard

alphabetic order, FALSE otherwise.

`s >= ss` yields TRUE if s is either identical with is or comes later in standard alphabetic order, FALSE otherwise.

`s <= ss` yields TRUE if s is either identical with ss or comes earlier in standard alphabetic order, FALSE otherwise.

`s IN ss` yields TRUE if s occurs as a substring of ss, FALSE if not.

`s NOTIN ss` yields FALSE if s occurs as a substring of ss, TRUE if not.

To give examples of these operators, we shall suppose that the value of s is the string 'ABRA', and that the value of ss is the string 'CADABRA'. Then

`print(ss(1),ss(4));`	yields C A
`print(s(1..2),s(2..4),s(2..2));`	yields AB BRA B
`print(s(1..0));`	yields the null string
`print(s(1..),s(2..),s(3..),s(4..));`	yields ABRA BRA RA A
`print(s(6..));`	yields the null string
`print(s(6));`	yields OM
`print(s+ss);`	yields ABRACADABRA
`print(3*s);`	yields ABRAABRAABRA
`print(s > ss,ss >s);`	yields FALSE TRUE
`print('AA' > 'A', 'A' > '') ;`	yields TRUE TRUE
`print('AA' < 'A', 'A' < '') ;`	yields FALSE FALSE
`print(s IN ss, ss IN s);`	yields TRUE FALSE

The unary string operators compute a value from a single string input s. These operators are

`#s` yields the number of characters in the string s.

`ABS s` here s must be a one-character string or an error results. If s is a single character, then ABS s returns the internal integer code for this character. Note that ABS and CHAR are are inverse operators.

`CHAR i` here i must be an integer which can be the internal code of some character c. If this is so, then CHAR i yields the single character c (i.e., a 1-character string). Otherwise, an error results. (The range of integer values used as character codes is implementation independent.)

The following table shows the way that the string extraction operators s(i), s(i..), and s(i..j) behave in various marginal cases.

Table 2.1. Behavior of String Operators in Marginal Cases

Operator	Condition	Effect
s(i)	i negative or zero	causes error
s(i)	i > #s	yields OM
s(i..)	i negative or zero	causes error
s(i..)	i = #s+1	returns null string
s(i..)	i > #s+1	causes error
s(i..j)	i negative or zero	causes error
s(i..j)	i > j+1	causes error
s(i..j)	j negative	causes error
s(i..j)	j > #s	causes error
s(i..j)	i = j+1	returns null string

To each string extraction operator there corresponds a string assignment operator which modifies the string section which the corresponding assignment operator would retrieve. These string assignments are indicated by writing either s(i), s(i..), or s(i..j) to the left of the assignment operator ':='. For example, if s is a string, we can modify the section of it extending from its second to its fourth character (inclusive) by writing

```
(1)              s(2..4) := x;
```

where x is any string. Note that x need not be a string of length 3, so that the assignment operation (1) can lengthen s (if x has length greater than 3) or shorten it (if x has length less than 3). Similar remarks apply to the string assignment operation

```
s(i..) := x;
```

which is treated exactly as if it read

```
s(i..#s) := x;
```

However, the right-hand side of the simple string assignment

```
s(i) := x;
```

must be a single character, or an error will result.

For examples of all this, suppose that s1,s2,...,s7 are seven variables, all having the string value 'ABRACADABRA' initially. Then the following assignments produce the indicated results.

```
s1(3..5) := 'XXX';          $ now s1 =  ABXXXADABRA
s2(3..4) := 'XXXXXX';       $ now s2 =  ABXXXXXXCADABRA
s3(3..4) := 'X';            $ now s3 =  ABXCADABRA
s4(3..4) := '';             $ now s4 =  ABCADABRA
s5(7..) := 'XXX';           $ now s5 =  ABRACAXXX
```

```
s6(7..) := '' ;          $ now s6  =   ABRACA
s7(1) := 'Y';            $ now s7  =   YBRACADABRA
```

To summarize, the three string assignment operators are:

`s(i) := x;` x must be a single character, and i must be an integer and lie between 1 and #s, otherwise an error results. This modifies the i-th character of s.

`s(i..j) := x;` i must be an integer at least equal to 1 and at most equal to j+1 or an error results. j must also be an integer, and cannot exceed s. The section of s between i and j is made equal to x, which may expand or contract s. Note that if i=j+1, x will be inserted into s immediately after its i-th position. The case i = #s+1, j = #s is legal, and adds x to the end of s.

`s(i..) := x;` this is treated exactly as if it read s(i..#s) := x. Thus i must be an integer which is at least 1 and at most #s+1.

As an example of the case i = #s+1, which is allowed, note that if s1 and s2 are both initially equal to 'ABC', then both the assignment

```
s1(4..3) := 'XXX' ;
```

and the assignment

```
s2(4..)  := 'XXX';
```

yield 'ABCXXX'.

2.3.4. Boolean Operators: AND, OR, IMPL, NOT

Boolean operators compute a boolean result from one or two input boolean quantities c, cc. That is, both the inputs of these operations and the results they produce must be one of the two possible boolean values TRUE and FALSE. These operations are generally used to combine results produced by prior comparisons or other tests, i.e. they typically appear in contexts such as

```
IF (i  > j AND j  > k) OR (k  > j AND j  > i) ...
```

The binary boolean operators supported by SETL are as follows:

c AND cc — yields TRUE if both c and cc are TRUE, FALSE otherwise.

c OR cc — yields TRUE if at least one of c and cc is TRUE, FALSE otherwise.

c IMPL cc — This is the 'logical implication' operator, and yields TRUE except when c is TRUE and cc is FALSE. That is, if either c is FALSE, or cc is TRUE, then c IMPL cc yields TRUE; but if c is TRUE and cc FALSE, then c IMPL cc yields FALSE.

The only unary boolean operator provided is

NOT C — yields the logical opposite of c, i.e., FALSE if c is TRUE, TRUE if c is FALSE.

In using these operations one will often make use of various well-known rules of logic like those called 'De Morgan's rules'. For example since

```
(NOT c) OR (NOT cc)
```

is TRUE if either c or cc is FALSE, but is FALSE if both c and cc are TRUE, it is equivalent to

```
NOT (c AND cc) .
```

Various other equivalences between boolean expressions are listed in the following table:

NOT (c OR cc)	is equivalent to	(NOT c) AND (NOT cc)
NOT (c IMPL cc)	is equivalent to	c AND (NOT cc)
c IMPL cc	is equivalent to	(NOT c) OR cc
NOT (NOT c)	is equivalent to	c

These and other related logical equivalences can often be used to simplify Boolean expressions that occur in programs. For example, since

```
c OR ((NOT c) AND cc)
```

is TRUE if and only if at least one of c and cc is TRUE, it simplifies to

c OR cc .

Thus, instead of writing

```
IF i  > j OR ((NOT i > j) AND k  > j) ...
```

in a program we can simplify this to

```
IF i  > j OR k  > j ...
```

Other useful relationships of this sort appear in Exercises 1 through 8 of Section 2.3.4.1.

2.3.4.1 Exercises

Boolean Equivalences

A tautology is a Boolean expression E which evaluates to TRUE no matter what Boolean values are given to the variables appearing in E. An equivalence is a statement of the form E1=E2 which evaluates to TRUE no matter what values are given to the variables appearing in it. Given any Boolean statement, we can easily write a program which substitutes values in all possible ways for the variables appearing in it, and this makes it easy to detect Boolean tautologies and equivalences. For example, since

```
{[x,y]: x IN {TRUE,FALSE},y IN {TRUE,FALSE}|(x AND y)/=(y AND x)}
```

evaluates to null, it follows that

```
(x AND y)=(y AND x)
```

is a universally valid Boolean equivalence. The following exercises list various tautologies and Boolean equivalences, which you are asked to prove either in this way or by appropriate mathematical reasoning.

Ex. 1 Prove the equivalence (A OR B)=(B OR A).

Ex. 2 Prove the equivalence ((A OR B) OR C)=(A OR (B OR C)), and also ((A AND B) AND C)=(A AND (B AND C)).

Ex. 3 Prove the equivalence (A AND A)=A, also (A OR A)=A.

Ex. 4 Prove the equivalence (A AND (B OR C))=((A AND B) OR (A AND C), also (A OR (B AND C))=(A OR B) AND (A OR C).

Ex. 5 Prove the equivalence (A OR ((NOT A) AND B))=A OR B).

Ex. 6 (De Morgan's Rules) Prove that (NOT (A AND B))=((NOT A) OR (NOT B)), also (NOT (A OR B))=((NOT A) AND (NOT B)).

Ex. 7 Prove that NOT(NOT A)=A. Using this fact and the results proved in Ex.6, show that

```
(A AND B)=(NOT((NOT A) OR (NOT B))), also that
(A OR B)=(NOT((NOT A) AND (NOT B))).
```

Ex. 8 Prove the following equivalences: (A AND TRUE)=A, (A AND FALSE)=FALSE, (A OR TRUE)=TRUE, (A OR FALSE)=A.

2.4 Set Operations and Setformers.

SETL provides several important kinds of set operators, of which the easiest to understand are the built-in, elementary set operations and the setformers discussed in Sec.2.4. We shall review these constructs in the present section; the even more important map operations are presented in Section X.

The binary set operations compute a result value from two inputs, one or both of which must be a set. These operations are as follows (in what follows, s and ss are always sets, while x can be an arbitrary value):

s + ss	computes the 'union' of two sets, i.e. the set of all objects which belong either to s or to ss.
s - ss	computes the 'difference' of two sets, i.e. the set of all objects which belong to s but not to ss.
s * ss	computes the 'intersection', or common part of two sets, i.e. the set of all objects which belong to both s and ss.
x IN s	tests x for membership in the set s. The value TRUE is produced if x is a member of s, FALSE otherwise.
x NOTIN s	tests x for nonmembership in the set s. The value TRUE is produced if x is not a member of s, FALSE otherwise.
s WITH x	produces a set whose members are the members of s, with x inserted (if x is not already a member of s)
s LESS x	produces a set whose members are the members of s, with x removed (if necessary, i.e., if x is a member of s)
s = ss	tests s and ss for equality, yielding TRUE if s and ss have exactly the same members, FALSE otherwise.
s/=ss	tests s and ss for inequality, yielding FALSE if s and ss have exactly the same members, TRUE otherwise.
s INCS ss	tests ss for inclusion within s, yielding TRUE if every member of ss is also a member of s, FALSE if ss has any member which is not also a member of s.
s SUBSET ss	tests s for inclusion within ss, yielding TRUE if every member of s is also a member of ss, FALSE if s has any member which is not also a member of ss.
n NPOW s	here the first argument n must be a nonnegative integer. This operation yields the collection of all subsets of s which contain exactly n elements. An error results if n is negative.
s NPOW n	here the first argument is a set and the second is an integer. This is equivalent to n NPOW s.

Examples of these binary set operators are

```
print({1,2}  + {'Tom','Dick'});          yields    {1 2 'Tom' 'Dick'}
print({}  + {1,2 , {} + {});             yields    {1 2}     {}

print({1,2,3} - {1,4}, {1,2,3} - {});    yields    {2 3}   {1 2 3}

print({1,2,3} - {3,1,2});                yields    {}

print({} -{1,2,3});                      yields    {}

print({1,2,3} * {2,5,3});                yields    {2 3}

print({1,2} * { 3,4 });                  yields    {}

print({} *  {3,4});                      yields    {}

print({{1},2,3} - {1,2,3});              yields    {{1}}

print({{1}, {2,3}} - {1,2,3});           yields    {{1},  {2 ,3}}

print(1 IN {1,2,3}  , {1} IN {1,2,3}  );  yields TRUE FALSE

print({}  IN  {}, {} IN  { {} });        yields FALSE TRUE

print(1 NOTIN  {1},  {}   NOTIN {});     yields FALSE TRUE

print({1,2,3}   WITH 5);                 yields  {1 2 3 5 }

print({1,2,3}   WITH 1);                 yields   {1 2 3}

print({1,2,3} LESS 1,  {1,2,3} LESS 4);  yields  {2 3}   {1 2 3}

print({1,2,3} = {3,2,1});                yields TRUE

print({} = [] , {}  = {{}}) ;            yields  FALSE FALSE

print({1,2} /= {2,1}, {1,2,2} /= {1,2});  yields FALSE FALSE

print(2 NPOW {1,2,3});                   yields { {1 2} {2 3} {1 3}}

print({1} INCS {}, {} INCS  {1});        yields  TRUE FALSE

print({1,2} INCS {1,2}) ;                yields  TRUE

print({2,2,2} SUBSET {1,2 }) ;           yields TRUE
```

Unary set operators compute a result value from a single set input s. The unary set operators are as follows:

#s yields the number of (distinct) elements of the set s

POW s yields the set of all subsets of s (which is also called

the 'power set' of s; hence the name POW)

RANDOM s yields a randomly selected element of s. Successive uses of RANDOM s will yield independently selected elements of s.

ARB s Yields an arbitrarily selected element of s. (Depending on the particular SETL implementation used, successive uses of ARB s may or may not yield the same element of s).

Examples of these unary operators are:

```
print(# {2}, #{2,2,2,2}) ;                    yields 1 1

print(# {1,2,3,4,1,2,3,4,40});                yields 5

print(POW {1,2}) ;                            yields {{} {1} {2} {1 2}
print(ARB {1 2,3}, ARB {1,2,3}) ;             yields 1 1
                                              (or possibly 2 2 or 3 3)
print(ARB {1,2,3}, ARB {3,1,2});              can yield 1 2 (even
                                              though
                                              {1,2,3} = {3,2,1}
                                              yields TRUE)

print(RANDOM {1,2,3}, RANDOM{1,2,3}) ;
                              (possibly) yields something like 2 1 2 3
```

Of course, the basic construct

```
{x1,x2,...,xk}
```

which forms a set by enumerating its elements explicitly is also a (multi-argument) set operator. The x1,x2,x3,...,xk appearing in this construct can be arbitrary expressions. As several of the preceding examples show, this construct can form a set of fewer than k elements. For example, if x has the value {1,2} and y the value {1}, then {x,y,x+y} is the two element set {1,2}.

As already noted, the set of all integers in the range from m to n (inclusive) can be written as

```
{n..m}
```

and the set of all integers n, n+k, n+2k, etc. up to m can be written

```
{n,n+k..m}
```

In this last form, the 'step' k can be negative, and n+k need not actually be a sum, but can be any arbitrary expression. For example,

```
print({3,6-1..10}) yields {3 5 7 9}
```

If the m in n..m is less than n, then the nullset results. Similar rules appy to {n,n+k..m}, for example

```
print({3,5..1}) yields { }

print({3,2..-3}) y {3 2 1 0 -1 -2 -3}

print({3,2..4}) yields { }

print({3,3..5}) results in an error.
```

See section 3.3.4 for additional details.

Many interesting mathematical relationships connect the set operators presented in this section. For example, the values of (s*s1) SUBSET s, and (s1+s2)*s3 = s1*s3+s2*s3 are always TRUE. Many other relationships of this sort appear in the exercises of Section 2.14.

2.4.1 Setformer Expressions

Sets are the basic data objects of SETL, and the language provides a number of ways of constructing sets. We have seen already in Sec.2.1.1 that constant sets are constructed by listing thier elements and enclosing the list between set brackets. More generally, sets can be constructed by enumerating their elements, be they constants, variables or expressions. For example, the set expression

```
{x, y, x+y, []}
```

describes a set whose components are the value of the variable x, the value of variable y, the expression (x+y) and the null tuple. Such sets constructed by enumeration can contain any number of expressions of any type.

In mathematics, the most powerful and general way of forming a set is simply to define it by stating a characteristic property of its elements. The standard mathematical notation for this is

(1) $\{x \mid C\}$

read 'the set of all x having the property C', or equivalently 'the set of all x such that C'. Any Boolean-valued expression can be used for C, for example we are allowed to write

(2) $\{x \mid x < 0\}$

whichh is read 'the set of all x such that x <0'. (As this example shows, the Boolean expression C of (1) will almost always depend on the variable x.)

SETL supports, and generalizes, a notation very close to (1). There is, however, one restriction which is always imposed. SETL is not only an abstract mathematical notation; it is also a programming language, which can be used to print out the actual value of any legal expression which it allows one to write Hence it works only with finite, not with infinite sets. This makes it necessary to impose a restriction on the way in which the notation (1) can be used, in order to prevent formation of obviously infinite sets like (2), which describes all of the negative numbers. This is done simply by insisting that the range of variation of the variable x in (1) be limited, in advance, by the condition that x should belong to some other finite object, e.g. some other set. That is, we allow, not exactly (1), but only the significantly more restricted construct

(3) $\{x \text{ IN } s \mid C\}$.

Then, since the set s used in (3) always has to be defined before (3) is evaluated, it follows that s must be finite; and then (3) must also designate a finite set whose list of elements can be calculated explicitly. In (3), we have the basic SETL setformer construct.

Several important generalizations of the construct (3) are used in mathematics and also allowed in SETL. Suppose, for example, that s is a set of numbers. Rather than simply forming the set (3), we may want to form a set of numbers obtained from (3) by applying some common transformation to all its elements, for example, by squaring them. To form this set, we are allowed to write

```
{x*x: x IN s | C}
```

which can be read: 'the set of all values x squared, for all x ranging over the set s such that C'. The general form of the more powerful kind of setformer is

(4) `{e: x IN s | C}`

In (4), e can be any expression, s any set valued expression, C any Boolean-valued expression. We can read (4) as 'the set of all values e, formed for those x in s for which C has the value TRUE'. Usually both e and C will depend on the value of x, i.e. on the various values of the members of s.

This reading of the notation (4) suggests a further generalization, which again is used in standard mathematics and is also legal in SETL. Specifically, there is no reason why in forming a set like (4) we should only allow one variable x to range over one set s. Instead, we can allow any number of variables to range over any number of sets. The notations

```
(5a)   {e: x IN s1, y IN s2 | C}
(5b)   {e: x IN s1, y IN s2, z IN s3 | C}
```

etc. express this more general constructions that this remark suggests. Note that (5b) can be read 'the set of all values e, formed for x ranging over s1, y (independently) ranging over s2, z ranging (again independently) over s3, but only in combinations x,y,z for which C has the value TRUE.'

Subsequently we will see that even further generalizations of the setformer constructs (3), (4), (5a), (5b), etc. are allowed. But, even as they stand, these constructs are extremely powerful, and we will now time to exhibit their power by giving a few interesting examples of their use. For this, we begin by considering the problem of printing out so-called prime numbers, for example all prime numbers in a given range, let us say the range {1...100}. We remind the reader that positive numbers like 6 = 2*3, 9 = 3*3, 4 = 2*2 which are the product of two smaller numbers, are called composite, and that numbers, larger than 1 which are not composite are called prime; examples of primes are 3,5,7,11,13,17... .

It is easy to express the set of all composite numbers up to 100 using a setformer (of type (5b)), namely as

(6) `{i*j: i IN {2...10}, j in {2...100} | i * j < 101} .`

Since the prime numbers we want are exactly the elements of {2...100} which do not belong to the set (6), we can print them out simply by writing

```
PRINT({2...100} - {i*j: i IN {2...10}, j in {2...50} | i*j < 100});
```

Sometimes the condition C appearing in (4), (5a), (5b), etc. is unnecessary. For example, given a set s of numbers we may simply want to form all the squares of numbers in s. In such cases one is simply allowed to drop the condition C, i.e. to write {e: x IN s}, read 'the set of all values e formed for x IN s'. Similarly, we can write

```
{e: x IN s1, y IN s2},

{e: x IN s1, y IN s2, z IN s3}, etc.
```

For example, we can write the set of all pairs x,y, where x ranges over s1 and y ranges over s2, as

```
{[x,y] : x in s1, y in s2}.
```

(In mathematics, this set is called the 'Cartesian product' of s1 and s2, afer Rene Descartes, the inventor of coordinate geometry.) Using these 'elided' setformers we can print the sets of primes considered above a bit more simply, for example we can print the primes up to 100 by writing

```
PRINT({2..100} - {i*j: i IN {2..10}, j in {2..50}})
```

Mathematicians who study prime numbers are often interested in primes having particular forms, for example primes p which are one more than a multiple of four, or three more, than a multiple of four. Since the set of all numbers (greater than 1) up to 100 which are one more (resp. three more) than a multiple of four can be expressed as

```
{4*n+1 : n in {0..24} | 4*n+1 < 101}
```
and
```
{4*n+3 : n in {0..24} | 4*n+3 < 101}
```

respectively, we can print the set of primes (up to 100) which are one more than a multiple of four by writing

```
PRINT({4*n+1: n IN {1..24} | 4*n+1 < 101}
      - {i*j: i IN {2..10}, j IN {2..50} | i*j < 101}) ;
```

and the corresponding set of primes which are three more than a multiple of four by writing

```
PRINT({4*n+3: n IN {0..24} | 4*n+3 < 101}
      - {i*j: i IN {2..10}, j in {2..50} | i*j < 101}) ;
```

2.4.2 Existential and Universal Quantifiers.

Very often, the key to a mathematical problem is to determine whether there exists any element x satisfying a given condition C, and the key to a programming problem lies in finding such an x if it exists. Using setformers, it is easy to express the condition that there should exist an x in s satisfying C: we have only to write

(7) {x IN s | C} /= {} .

Moreover, if the condition (7) is satisfied, we can easily find such an x, simply by writing

(8) ARB {x IN s | C} .

Since the test (7) is so important and common, a special abbreviation is provided for it, namely

(9) EXISTS x IN s | C .

This is a boolean-valued expression, yielding exactly the same value as (7). Moreover, if it yields the value TRUE, it will set x to the value of (8), i.e. to some value satisfying C. If (7) is false, then the variable x in (8) gets

value OM.

As in a setformer, the s in (9) can be an arbitrary set-valued expression, while C can be an arbitrary boolean valued expression.

Generalizations of (9) corresponding to the generalized setformers (5a), (5b) are allowed. Specifically, one can write

(10a) EXISTS x IN s1, y IN s2 | C
(10b) EXISTS x IN s1, y IN s2, z in s3 | C

etc., where s1,s2,...are arbitrary set-valued expressions and C a Boolean expression. The constructs (10a), (10b), ec. search the set of all x in s1, y in s2, ... for values satisfying the condition C. If such values are found, then (10a) (or (10b)) yields the value TRUE and the variables x,y,... are set to these values. Otherwise (10a) (or (10b)) yields the value FALSE and x,y,.. get indeterminate values.

The constructs (9), (10a) (10b) etc. are called existential quantifiers.

The existential quantifier allows us to express naturally the common query : does there exist an object in a certain collection, which satisfies a given criterion ? A related query, which is also very common in programming contexts, is the following : do ALL the objects in a collection satisfy some stated criterion ? Such queries are expressed in SETL by means of constructs such as the following:

(11a) FORALL x IN s | C
(11b) FORALL x IN s1, y IN s2 | C
(11c) FORALL x IN s1, y IN s2, z in s3 | C

which make use of the keyword 'FORALL' . These constructs which are called universal quantifiers, are closely related to existential quantifiers. The three cases just given are equivalent to:

(12a) NOT EXISTS x IN s | (NOT C)
(12b) NOT EXISTS x IN s1, y IN s2 | (NOT C)

```
(12c)     NOT EXISTS x IN s1, y IN s2, z IN s3 | (NOT C)
```

respectively. For example, (11c) searches the set of all x in s1, y in s2, z in s3 for values such that the condition C takes on the value FALSE. If none exists then (11c) returns the value TRUE (and the variables x,y,z take the value OM). HHowever, if values satisfying C exist, then (11c) returns the value FALSE (and the variables x,y,z take on values (in s1, s2, and s3 respectively) fulfilling the condition C).

By using quantifiers we can write a simpler and more readable setformer representing the set of all primes up to 100. Specifically, an integer n is prime if there exists no smaller integer m (other than 1) which divides n evenly, i.e. such that n MOD m = 0. Hence

```
PRINT({n in {2..100} | NOT EXISTS m in {2..n-1} | n MOD m = 0});
```

will print the set of primes up to 100. Similarly,

```
PRINT({n in {2..100} | ((NOT EXISTS m in {2..n-1} | n MOD m = 0)
                                    AND (n-1) MOD 4 = 0));
```

will print all the primes up to 100 which are one more than a multiple of four, while

```
PRINT({n in {2..100} | ((NOT EXISTS m in {2..n-1} | n MOD m = 0)
                                  AND (n-3) MOD 4 = 0)});
```

will print the set of all primes up to 100 which are three more than a multiple of four.

As we have said, the existential quantifier (9) returns exactly the same value as the expression (7). However, the quantifier calculates this value more efficiently than (7) would, since to evaluate (9) the SETL system will search systematically through the elements of s but stop searching and return the value TRUE as soon as an x satisfying C has been found, whereas to evaluate (7) it would always search through the <u>whole</u> of s building up the set {x in S | C}, and only test it for nullity after it had been evaluated fully. This distinction becomes particularly important if evaluation of the boolean condition C causes side effects, since in this case evaluation of the two expressions (8) and (9) will have different cumulative side effects. Similar remarks apply to universal quantifiers (11), (11a), and (11b).

<u>A remark on bound variables in compound setformers and quantifiers</u>

The variables x, y, z occurring in (9), (10a-b),(11a-c), and (12a-c) are called <u>bound variables</u>, since the quantifiers in which they appear cause them to be iterated over some set. Quantifiers (or setformers) such as (10a-b), (11b-c), or (12b-c) involving more than one bound variable cause multiple iterations, e.g. in evaluating (10a) x is given successive values from the set s1, and then for each of these values of x , y is given all possible values from s2. For this reason, the expression s2 in (10a) is allowed to depend on the bond variable x, but s1 must be independent of y. Similarly, in 10b), s3 can depend on both x and y, s2 can depend on x but

not y, and sl cannot depend on either x or y. Similar rules apply to universal quantifiers and to setformers.

2.4.3. Some illustrative one-statement programs.

Thus far we have introduced only a few of the facilities which the SETL language makes available. Only one or two of the commands available to the programmer have been described yet, so that we cannot yet show any substantial programs. However, the mechanisms that have been described are powerful enough to allow various interesting single-statement programs to be written. In this section, we collect a few such progams.

a. More about prime numbers.

As noted in the preceding section, an integer is called prime if it is not evenly divisible by any smaller (positive) integer other than 1.

To form the set of all prime numbers up to 100 we can use the one-line program given in the preceding section, which simply prints a setformer:

```
PRINT({n in 2..100| NOT EXISTS m in 2..n-1| (n MOD m) = 0});
```

The output of this single-statement program is

```
{2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97}
```

Note however that since sets are not ordered the elements of this set can be printed in any arbitrary order. The actual order used (which has no logical significance) will depend on the particular SETL implementation which o ne is using.

Mathematicians who study prime numbers are sometimes interested in find not all the primes in a given range, but only those which have various special properties. For example, a prime n is said to belong to a prime pair if both n and n+2 are primes. (Note that, since all primes except 2 are odd, we cannot expect both n and n+1 to be prime, because if n is a prime then n+1 will be even, hence not a prime.) To find all prime pairs up to 100 we can simply write

```
PRINT({n IN  {2..100} |
              (NOT EXISTS m in {2..n-1}   | (n MOD m) = 0)
         and (NOT EXISTS m in {2..n+1 } | ((n+2) MOD m) = 0)});
```

The output of this program is

```
{3  5  11  17  29  41  59  71}
```

indicating that the only such twin-prime pairs are

[3,5], [5,7], [11,13], [17,19], [29,31], [41,43], [59,61], [71,73],

Sometimes one is interested in primes which satisfy particular quadratic equations, for example primes n of the form n = k**2+1. Since if n is not larger than 100, any integer k solving this equation would have to be smaller than 10, we can find all the primes of this form just by writing

```
PRINT({n in {2..100} | (NOT EXISTS m IN {2..n-1} | (n MOD m) = 0)
                        and (EXISTS k IN 0..10 | n = k*k+1)});
```

Similarly, to find al the primes up to 100 which have the form 2k**2+3 we can write

```
PRINT({n in {2..100} -| (NOT EXISTS m IN {2..n-1} | (n MOD m) = 0)
                        and (EXISTS k IN {0..10}  | n = 2*k*k+3)});
```

the output of the first of these programs is

{2 5 17 37}

and the output of the second program is

{3 5 11 53}

b. Integer right triangles.

The famous theorem of Pythagoras states that the length h of the hypotenuse of a right triangle and the lengths a and b of its two sides are related by the equation a ** 2 + b ** 2 = h ** 2. Whole-number solutions of this equation are useful to people who make up elementary mathematics exams and want to invent problems that have whole number answers. Examples of such 'integer right triangles' are 3,4,5 and 5,12,13. The following single-statement program finds all integer right triangles a,b,h for which a is less than b and both are less than 30. We let b range over the set {1..30}, and a range over the set {1..b-1}. To find if a*a + b*b is a perfect square, we simply search for an integer h qhose square is equal to that sum. The possible range of h is from 1 to a+b. (Approximately. Can you give a more precise range for it ?).

Note that we eliminate all triangles for which a and b have a common divisor, since these are simple multiples of smaller integer right triangles.

```
PRINT({ [a,b,h] : b IN {1..30}, a IN {1..b-1} |
       (EXISTS h IN {2..a+b} | (a*a+b*b = h*h)) and
    NOT EXISTS c IN {2..b-1} | ((b MOD c) = 0 and (a MOD c) = 0)});
```

The output of this program is

{[3 4 5] [5 12 13] [8 15 17] [20 21 29] [7 24 25]}.

It is not hard to prove mathematically that there exist infinitely many different integer right triangles.

2.5 Tuple Operations and Tuple Formers

We have mentioned repeatedly that sets are unordered and can never have duplicate or undefined members; tuples are ordered and can have both duplicate and undefined components. For example,

```
[1,0,1,0,OM,OM,1,0]
```

is a perfectly legal tuple; its first, third, and seventh components are all 1, while its fifth and sixth components are undefined. In spite of this very fundamental difference between sets and tuples, the binary and unary operators on tuples which SETL provides are similar to corresponding set operators. In addition, tuple formers that construct tuples in the same manner that set formers build sets, exist with a similar sybtax. In fact, all set forming expressions can be transformed into tuple forming expressions, by replacing the set brackets with tuple brackets.

2.5.1 Binary Tuple Operators

Binary tuple operators compute a result value from two inputs, one or both of which must be a tuple. The binary tuple operators are as follows (in what follows, t and tt are always tuples, while x can be an arbitrary value):

`t + tt`	concatenates tt to the end of t.
`n * t`	here, n must be an integer. This forms n copies of t and concatenates them end to end, to form a tuple n times as long as t. If n = 0, then the null tuple (i.e. []) is obtained, if n < 0, an error results.
`t * n`	if n is an integer, this is equivalent to n*t
`x IN t`	yields TRUE if x equals one of the components of t; FALSE otherwise.
`x NOTIN t`	yields FALSE if x equals one of the components of t; TRUE otherwise.
`t WITH x`	yields a new tuple identical to t except that x is appended to it as an additional final component
`t = tt`	yields TRUE if all components of t are identical to the corresponding components of tt, FALSE otherwise.
`t /= tt`	yields TRUE if some component of t differs from the corresponding component of tt, FALSE otherwise.

It should be noted that a tuple is considered to extend from its first component to its last defined component, i.e., its last component differen from OM. That is, all tuples are regarded as ending with an indefinitel,

long sequence of OM components, but when a tuple is printed only its non-OM components are shown. For example,

```
[OM,OM,OM,OM]     is equivalent to        []

[1,OM,2,OM]       is equivalent to        [1,OM,2]

[1,OM] WITH OM    is equivalent to        [1]
```

Some examples of the binary tuple operators are:

```
print([1,2] + [3,4])              yields  [1,2,3,4]

print([1,2] WITH [3,4]);          yields  [1 2 [3 4]]

print(2*[1,2], [1,3]*2);          yields  [1 2 1 2]  [1 3 1 3]

print(1 IN [1,2,3],  [1,2] IN [1,2,3]);
                                  yields  TRUE FALSE

print(OM IN [1,2,3], OM IN [1,OM,3]);
                                  yields FALSE TRUE

print([1,2]=[2,1], [1,2,1,2] = [1,21,2]);
                                  yields FALSE FALSE

print([1,2,1,2]/= [1,2,1,2,1], [1,1]/=[1,1,1], [1]/=[1,OM]);
                                  yields  TRUE TRUE FALSE

print({} /= []);                  yields TRUE
```

2.5.2 Unary Tuple Operators

Unary tuple operators produce a value from a single tuple operand. The unary tuple operators are:

#t	yields the index of the last non-OM component of t
RANDOM t	yields a component of t picked at random from its first to its last non-OM component. All components, including OM components in this range, have an equal chance to be picked. Note that successive uses of RANDOM t will generally yield different, independently chosen random components.

The following are examples of the unary tuple operators.

```
print(#[3], #[], #[1,OM]);                    yields  1 0 1

print(#[1,OM], #[OM,1], #[1,1,1]);            yields  1 2 3

print(#[1,OM,OM,OM,OM,1]);                      yields  6
```

```
print(#[1,OM,OM,OM,OM]);                          yields 1

print(#[1,2,3,4], #[1,2,[3,4]], #[[1,2,3,4]]);  yields 4 3 1

print(RANDOM [1,2,3], RANDOM [1,2,3], RANDOM [1,2,3], RANDOM [1,2,3]);

          (probably) yields something like 2 1 2 3
```

2.5.3 Other Tuple Operators

As for sets, so for tuples the construct

```
[x1,x2,...,xk]
```

which forms a tuple by enumerating its elements explicitly, is also a (multi-argument) tuple operator. As should be obvious, the various xj appearing in this construct can be arbitrary expressions. If some of the x appearing at the end of this construct evaluate to OM, then a tuple o length less than k will be formed. For example, if t has the value [1,OM,OM,2], then

```
[t(4), t(3), t(2), t(3)]
```

forms the tuple [2, OM, OM, OM], i.e. the tuple [2], whose length is o course 1.

The tuple of integers ranging from m to n (inclusive) can be written a

```
[n...m]
```

and the tuple of integers n, n+k, n+2k, etc. up to m can be written

```
[n, n+k,...,m].
```

In this last form, the 'step' k can be positive (producing an ascending sequence) or negative (producing a descending sequence). The quantity n+k need not actually be a sum, but can any integer-valued expression. If the m in [n...m] is less than n, then the null tuple results. Similar rules apply to [n, n+k,...,m]. For example,

```
print([3,5,...,1]);       yields       []
print([3,2,...,-3];       yields       [3 2 1 0 -1 -2 -3]
print([3,2,...,4]);       yields       []
print([3,3,...,5]);       yields       []
```

Tuple indexing, 'slice' and assignment operators, which resemble th string slice and assignment operators described in Section 2.3.3, ar provided. The indexing and slice operators are as follows (we assume as before that t designates a tuple):

```
t(i)     yields the i-th component of the tuple t.
```

If i is zero or less, an error results;
if i exceeds the index of the last non-OM
component of t, then t(i) yields OM.

t(i..j) yields the section or 'slice' of t extending from its i-th through its j-th components, inclusive. If i is zero or negative, or if i exceeds j+1, an error results.
If i = j+1, then t(i...j) always yields the null tuple. If i exceeds the last non-OM component of t, then a null tuple is returned.

t(i..) yields the section or 'slice' of t extending from its i-th through its last non-OM component, inclusive. This operator is equivalent to t(i..#t). Thus if i is zero or negative, or if i exceeds #t+1, an error results.
If i = #t+1, then t(i..) yields the null tuple.

To give examples of these operators, we assume that t is the tuple [10,OM,30,OM,50,OM,70]. Then:

```
print(t(1), t(2), t(3));      yields   10  OM  30

print(t(7), t(8));            yields   70  OM

print(t(2..5), t(2..6));      yields   [OM 30 OM 50] [OM 30 OM 50]

print(t(2..8));               yields   [OM 30 OM 50 OM 70]

print(t(3..2));               yields   []

print(t(8..11));              yields   []

print(t(3..), t(8...)) ;      yields   [30 OM 50 OM 70] []

print(t(9..));                results in an error
```

It should also be noted that if the ith component of t is itself a tuple or a string, then further indexing of t(i) is possible. Suppose, for example, that t is the following tuple of tuples of strings:

```
[ ['Tom','Dick','Harry'], ['Peter','Paul','Mary'], ['Mutt','Jeff']]
```

Then:

```
t(2)                          yields  [Peter Paul Mary]
t(2)(3)                       yields  Mary
t(2)(3)(1)                    yields  M
t(2..3)                       yields  [[Peter Paul Mary] [Mutt Jeff]]
t(2..3)(2..)                  yields  [[Mutt Jeff]]
t(2..3)(2..)(1)               yields  [Mutt Jeff]
t(2..3)(2..)(1)(2..)          yields  [Jeff]
t(2..3)(2..)(1)(2..)(1)(2)    yields  e
```

Similar constructs involving map assignments are allowed; see Section 2.12.

The tuple assignment operators are as follows (we assume as before that the values of t and tt are tuples):

`t(i) := x ;` modifies the i-th component of the tuple t, setting it equal to the value of x. If i is zero or negative, an error results. If i exceeds the index of the last non-OM component of t, then t will be extended with as many OM components as necessary, and then its i-th component will be set equal to x. (Therefore the assignment t(i) := x can increase the length of t by any amount up to i)

`t(i..j) := tt;` modifies the section of t extending from its i-th through its j-th component, setting it equal to tt. If i is zero or negative, or if i exceeds j+1, an error results. If i = j+1, then tt will be inserted into t immediately following position i. If i exceeds the index of the last non-OM component of t, then t will be extended with as many OM components as necessary, and then tt will be appended.

`t(i..) := tt ;` this assignment is equivalent to t(i..#t) := tt. Thus it modifies the section of t extending from its i-th component to its last non-OM component, setting it equal to tt. If i is zero or negative, or if i exceeds #t+1, an error results. If i = #t+1, then tt is appended to the end of t.

To give examples of these operators, suppose that t1, t2, ..., t22 all have the value [1,2,3,OM,OM,6]. Then

```
t1(2) := OM ;  $ now t1 = [1 OM 3 OM OM 6]

t2(4) := 40 ;  $ now t2 = [1 2 3 40 OM 6]

t3(8) := 70 ;  $ now t3 = [1 2 3 OM OM 6 OM 8]

t4(9) := OM ;  $ now t4 = [1 2 3 OM OM 6]

t5(2..4):= [OM 30 40] ;  $ now t5 = [1 OM 30 40 OM 6]

t6(2..2) := [20] ;  $ now t6 = [1 20 3 OM OM 6]

t7(2) := 20 ;  $ now t7 = [1 20 3 OM OM 6]

t8(2) := [20] ;  $ now t8 = [1 20 3 OM OM 6]

t9(2..2) := 20 ;  $ results in an error
```

```
t10(2..1):= [20 OM 30] ;  $ now t10 = [1 2 20 OM 30 3 OM OM 6]

t11(6..5):= [20 OM 30] ;  $ now t11 = [1 2 3 OM OM 6 20 OM 30 ]

t12(1..0):= [20 OM 30] ;  $ now t12 = [1 20 OM 30 2 3 OM OM 6]

t13(8..9):= [20 OM 30] ;  $ now t13 = [1 2 3 OM OM 6 OM 20 OM 30]

t14(5..5):= [20 OM 30] ;  $ now t14 = [1 2 3 OM 20 OM 30 6]

t15(5..5):= [20 OM OM] ;  $ now t15 = [1 2 3 OM 20 6]

t16(4..5):= [];  $ now t16 = [1 2 3 6]

t17(2..3):= [20] ;  $ now t17 = [1 20 OM OM 6]

t18(2..4):= [20] ;  $ now t18 = [1 20 OM 6]

t19(6..) := [] ;  $ now t19 = [1 2 3]

t20(5..):= [50 60 70 80] ;  $ now t20 = [1 2 3 OM 50 60 70 80]

t21(7..):= [50 60 OM 80] ;  $ now t21 = [1 2 3 OM OM 6 50 60 OM 80]

t22(8..):= [20 OM 30] ;  $ results in an error
```

Repeatedly indexed tuple (and map) assignments such as

```
t(i)(j..k)(l) := tt;
```

are possible in some cases; see Section 2.12 for a general discussion of these assignments.

2.6 Tuple Formers. Simple Tuple and String Iterators.

The construct

(1) `[e: x IN s | C]`

read 'the tuple of all values assumed by the expression e as x ranges over the elements of s for which the condition C has value TRUE' is similar to the setformer

(2) `{e: x IN s | C}`,

(see Section X) except that (2) eliminates duplicates and builds a set, whereas 1) builds a tuple and does not eliminate duplicates. The order in which the components of the tuple (1) are arranged is determined by the order in which iteration proceeds over the elements x of the set s.

As in the case of setformers, the condition C appearing in (1) need not appear, i.e. one can write

(1) `[e: x IN s]`

read 'the tuple of all values assumed by the expression e as x ranges over all the elements of s'. Moreover, multiple iterations can be used in a tuple formers, i.e. constructs like

(3a) [e: x IN s1, y IN s2]
(3b) [e: x IN s1, y IN s2, z in s3]

etc., are allowed. Again, the order in which the components of (3a) or (3b) are arranged depends on the order in which iteration proceeds over the elements of s1, s2, etc. However, in (3a) and (3b) a complete iteration over s2 will always be made each time the variable x advances from one element of s1 to the next, and in (3b) a complete iteration over c3 will always take place each time the variale y advances from one element of s2 to the next.

If the e in (4) is simply x, then it can be elided, i.e. we can simply write

[x IN s | C]

read 'the tuple of all x IN s for which the condition C evaluates to TRUE'. It is even possible to elide both e and C, thereby writing

[x IN s]

this simply arranges the elements of the set s in (arbitrary) order as a tuple. Notice that s itself could be a tuple, in which case [x in s] is simply another copy of the tuple s. Similar elisions are allowed for setformers.

As noted in Section 2.4.2, the 'iterator' x IN s appearing in such constructs as the set former

(4) {e: x IN s | C}

and the existential quantifier

(5) ...EXISTS x IN s | C..

iterates over the elements of s, assigning each one of them in turn as the value of x, until the iteration terminates, either because (as in (4)) all elements of s have been processed, or because (as in (5)) an element x of s satisfying the condition C has been found. Since iterative constructions and searches of this kind are quite useful, corresponding iterators over tuples and strings are also provided. If t is a tuple, then the iterator

-x IN t-, which can be used in such contexts as

(4a) {e: x IN t | C}

and

(4b) ...EXISTS x IN t | C..

iterates over the components of t, <u>in order</u>, from its first component to its

last non-OM component, assigning each component in turn as the value of the variable x, until the iteration terminates for one of the two possible reasons stated above. The iteration advances over all components, including OM components, in turn, but components not satisfying the Boolean condition C appearing in (4a) and (4b) are bypassed. We emphasize that, even though the corresponding set iterator, e.g.

```
...EXISTS x IN s | C
```

can iterate over the elements of the set s in some unpredictable, arbitrary order, the tuple iterator (4b) always iterates over the components of t in a known order, namely from first component to last. Therefore, if the existential search (4b) finds any component x of t satisfying the condition C, it will always find the leftmost such component, which will become the value of x.

We can iterate over the successive characters of a string in similar fashion. If in (4a) t is a string, then (4a) iterates over its characters, in order, from its first character to its last, assigning each character in turn as the value of the variable x, until the iteration terminates for one of the two possible reasons stated above. Characters not satisfying the condition C appearing in (4a) are bypassed. Similar remarks apply to the setformer (4a) and to universal quantifiers which iterate over strings and tuples.

Note, as an easy application of all this, that the set s of all distinct components of a tuple t can be formed by writing

```
{x IN t}.
```

If t is a string, this same expression will form the set of all its distinct characters.

For a more general account of the iterator forms usable in setformers, tuple formers, compound operators, and FOR-loops, see Section 3.3.

By writing the iterator

```
x IN [M..N]
```

as part of a set former or quantifier we can cause x to be iterated over all the integers of the numerical range M through N inclusive in order. Similarly, by writing the iterator

```
x IN [M,M+k..N]
```

we cause x to be iterated over integers lying between M and N, starting with M and proceeding by steps of k. This iteration will proceed either in increasing or in decreasing order, depending on whether k is positive or negative. (If k = 0, the iteration will be terminated as soon as it is attempted.) For example, to find all the vowels in a string which are followed by other vowels and print the corresponding set of all double vowels or 'dipthongs', we can simply write

```
print({s(i..i+1): i IN 1..#s-1 | s(i) IN vowels AND s(i+1) IN vowels});
```

(where the variable -vowels- must first be assigned the value {'a','e','i','o','u','y'}. Similarly, to find the set of all places in a tuple of integers at which the sign of its component changes from + to -, we can simply write

```
print({i IN [1..#t-1] | t(i) > 0 AND t(i+1) < 0}).
```

2.7 Map operations

Sets of a somewhat special kind, namely sets all of whose elements are pairs (that is, all of whose elements are tuples of length 2) have a very special importance in SETL because they can be used to record associations between pairs of objects. Sets of this kind are called maps, and the most significant operators of SETL, its so-called map operators, apply only to such sets. In this section, we will describe these operators and review their use.

2.7.1 The image-set operator f{x} and the image operator f(x).

Suppose that f is a map, i.e. a set of pairs

(1) {[x1,y1], [x2,y2], ..., [xk,yk]} .

Then f{x}, called the image set of f at the point x, is defined to be the set of all second components of pairs in f whose first component is x. Using the standard set former, we can write this set as

(2) {y(2): y in f | y(1) = x}.

The significance of this operation lies in the fact that, if we regard f as representing a certain abstract relationship R, then f{x} is precisely the set of all elements which stand in the relationship R to the object x.

Suppose, for example, that f contains the pair [s,c] if and only if s is a student in a particular school and c is a course in which s is registered. Then f{s} designates the set of all courses in which student s is registered. Suppose next that g is another map, which contains the pair [c,s] if and only if f contains the pair [s,c]. (This map is called the inverse of the map f.) Then for each course c, g{c} is the set of all students registered in the course.

For a still more specific example, suppose that f is the map

(3) {['Jones','Tom'], ['Khalid','Leila'], ['Smith','Mary'],
['Khalid','Fatima']}

Then:

f{'Jones'} is {'Tom'}; f{'Smith'} is {'Mary'}; f{'Khalid'} is {'Leila','Fatima'}

moreover

f{'Chang'} is the nullset ({})

Since no pair beginning with 'Chang' is present in the map (3).

Note that the DOMAIN of f, namely the set of all first components of pairs in f, is also the set of all x for which f{x} is different from {}, and that the RANGE of f, namely the set of all second components of pairs in

f, is also the set of all y which belong to at least one set of the form f{x}.

2.7.2 The single-valued image operator f(x)

If the image set f{x} contains exactly one element y, that is, if f{x} is {y}, then we can also write this element y simply as f(x) (rather than as ARB f{x}) The quantity f(x) is called the image (or sometimes, for additional emphasis, the single-valued image), of the element x under the map f, and we say that the map f sends x into f(x). If x is not in the domain of f, so that f{x} is empty, or if f{x} contains more than one element, then f(x) yields the value -undefined- or OM.

This last rule can be understood as follows. If, as before, we regard f as representing an abstract relationship R, then f(x) represents the unique element y which stands in the relationship R to x. If x is not in DOMAIN f, then f(x) is obviously undefined, since no element stands in the relationship R to x. If f{x} contains more than one element, then f(x) is still undefined, since we cannot tell which one of the several elements of f{x} the expressions f(x) is supposed to represent. We can only speak of the element standing in the relationship R to x if f{x} contains exactly one element; thus the case in which f(x) gives a non-OM value.

For an example of all this, suppose once more that f is the map (3). Then

```
f('Jones')  is 'Tom';  f('Smith')  is 'Mary';
f('Chang')  is OM, since 'Chang' is not in the domain of f;
f('Khalid') is OM, since f{'Khalid'} is a set
                containing more than one element.
```

A map f is said is called single-valued at x if f(x) is defined, but is called multiple-valued at x if f{x} contains more than one element. The map f is said to be a single-valued map (or simply to be single valued) if it is single-valued at each element x of its domain.

Note that maps are also sets (namely sets all of whose elements are tuples of length 2), so that all set operations also apply to maps. In particular, we can form the union, intersection, and difference of maps, add elements to and subtract elements from a map using the WITH and LESS operators, evaluate f where f is a map, etc. . Note that if f and g are both maps, then f+g, f*g, and f-g are also maps since every element of any one of these sets will be a pair; the same remark applies to f LESS z for any z. Moreover, if f is a map and z is known to be a pair, then f WITH z is still a map sice all its elements are pairs. For example, if f is the map (3) and we let f2 be 'f WITH ['Jones','Sue'], then f2 is still a map, moreover f2{'Jones'} is {'Tom','Sue'}, and f2('Jones') is OM.

SETL allows us, not only to evaluate expressions like f{x} and f(x), but also to use such expressions as assignment targets. If the value of f is a map, the map assignment

(4) f(x) := y ;

is always legal. The effect of this assignment is to modify f, and, as the notation (4) is intended to suggest, to modify it in such a way as to cause the value of f(x) to be y if f(x) is evaluated immediately after the assignment (4) is executed. This is done by modifying f as follows:

(a) First, all pairs [x,z] whose first component is x are removed from f. (This has the effect of removing x from DOMAIN f).

(b) Next (if y has a value other than OM), the single pair [x,y] is inserted into f. Thus f will contain exactly one pair [x,y] whose first component is x, guaranteeing that f(x) will evaluate to y.

(c) However, if y has the value OM, then only step (a), but not step (b), is performed. In this case x will simply have been removed from DOMAIN f, guaranteeing that f(x) will evaluate to OM.

Rules (a), (b), and (c) tell us that if y /= OM, then (4) has exactly the same effect as the assignment

```
(5a)      f := {z: z in f | z(1) /= x} WITH [x,y];
```

while if y = OM, then (4) has the same effect as the assignment

```
(5b)      f := {z: z in f | z(1) /= x}.
```

The intuitive significance of the assignment (4) can be explained as follows: it directs us to drop any prior association to the element x that is recorded in f, and then to associate x with y (for which we insert the pair [x,y] into f if y /= OM, but simply leave x without any association if y = OM). This is exactly the effect of steps (a-c).

For examples of all this, suppose again that f is the map (3), and that we first perform the assignment

```
f('Jones') := 'Thomas';
```

This changes f to

```
{['Jones','Thomas'], ['Khalid','Leila'], ['Smith','Mary'], ['Khalid',
'Fatima']}
```

Suppose that the assignment

```
f('Chang') := 'Zhong-Tien' ;
```

is performed next. In this case, no pairs need to be removed from f, but one pair is added, changing f to

```
{['Jones','Thomas'], ['Khalid','Leila'],['Smith','Mary'],
                     ['Chang','Zhong-Tien'], ['Khalid','Fatima']}
```

Next, suppose that the assignment

```
f('Cohen') := OM ;
```

is performed. This will simply remove all pairs with first component 'Cohen' from f; but since there are none such, it will actually leave f unchanged. After this, suppose that the assignment

```
f('Khalid') := 'Nuri' ;
```

is performed. This removes the pairs ['Khalid','Leila'] and ['Khalid','Fatima'] from f, and gives f the value

```
{['Jones','Thomas'], ['Smith','Mary'], ['Khalid','Nuri'],
                                ['Chang','Zhong-Tien']}
```

Assignments of the form (3), which change the element y associated with an element x, are generally used for one of three purposes:

(i) to update an attribute f(x) of x;
(ii) to define an attribute of x which has previously been undefined;
(iii) to drop an attribute f(x) that is no longer needed, which we do by executing f(x) := OM.

Suppose, for example, that f is being used to keep track of the number of times that each word x has been seen in a body of text that is being scanned. On encountering a word, we test to see if it has been seen before; if so, we simply increment its count. Otherwise, we must initialize its count attribute, which will be undefined, to the value 1. This is done by the following code, which uses several map assignment operations.

```
IF f(x) = OM THEN          $ word is new
   f(x) := 1;              $ establish initial count for new word
ELSE
   f(x) := f(x)+1;         $ increment count of word previously seen
END IF;
```

Note that a map assignment f(x) := y begins (see (a) above) by attempting to remove a certain set of pairs from f, which assumes that f is already a map. Hence the operation f(x) := y (like the operations y := f(x) and y := f{x}) can only be applied if f is already a map. The question then arises as to how to initialize a map f. This can be done in one of two ways:

(i) If f is initially supposed to be the ('everywhere undefined') map whose domain is null (so that initially f(x) = OM for all x and f{x} = {} for all x), we simply put

```
f := {} ;
```

This makes f the everywhere undefined map with null domain and null range.

(ii) A map value can be built up directly using a setformer, providing that all elements of the set which is formed are pairs. For example, we can write

```
f := {[x,#x]: x in {'Tom','Dick','Harry'}};
```

this makes f a map with domain {'Tom','Dick','Harry'}, and f maps each element x in its domain into the length of x.

The multivalued map assignment

```
(6)          f{x} := y ;
```

is also legal in SETL. As the notation (6) suggests, this assignment modifies f in such a way as to cause the value f{x} to be y if f{x} is evaluated immediately after the assignment (4) is executed. It follows that (6) makes no sense, and will generate an error, if the value of y is not a set.

The multivalued map assignment (6) is performed as follows.

(a) We first check that f is a map (i.e. a set consisting of pairs only), and that y is a set. If either of these conditions is violated, an error is generated.

(b) All pairs x,y whose first component is x are removed from x. (This has the effect of removing x from DOMAIN f.)

(c) After this, the set of all pairs x,z, for all y, is added to f. This guarantees that f{x} will evaluate to y.

These rules tell us that (6) has exactly the same effect as the assignment

```
(7)     f := {u: u in f |  u(1) /= x} + {[x,z]: z in y}.
```

Note therefore that if y /= OM, (4) has exactly
the same effect as the map assignment

```
(8a)          f{x} := {y} ;
```

while if y = OM, then the effect of (4) is exactly that of

```
(8b)          f{x} := {} ;
```

The value (5b) given to f by either f(x) := OM or by f{x} := {} can also be written in another form, namely as the expression

```
(9)          f LESSF x
```

which occasionally is more convenient. Note that (9), like the map assignment operators, applies only to maps, and will generate an error if applied to set f which contains any non-pair elements.

As an example of all this, suppose again that f is the map

```
(3')  {['Jones','Tom'], ['Khalid','Leila'], ['Smith','Mary'],
                                  ['Khalid', 'Fatima']}
```

Then the assignment

```
        f{'Khalid'} := f{'Khalid'} WITH 'Omar' ;
```

gives f the value

```
 {['Jones','Tom'], ['Khalid','Leila'], ['Khalid','Omar'],
                        ['Smith','Mary', ['Khalid','Fatima'] }
```

If we subsequently execute the assignment

```
        f{'Jones'} := {} ;
```

then f will take on the value

```
 {['Khalid','Leila'], ['Khalid','Omar'], ['Smith','Mary'],
                              ['Khalid','Fatima'] }
```

Along with the general set former construct, the map operations f(x), f{x}, f(x):=y, and f{x}:=y are the most characteristic and important operations of the SETL language. Their importance derives from the fact that they allow arbitrary objects x to appear as 'indices', i.e. any object can appear as the x in a construct like f(x) or f(x):=y. Of course, other lower level programming languages, such as PL/1, PASCAL, and Ada, support constructs with exactly this syntax and with a very similar intended use. However, in these other languages, an f used in this way must be an 'array' (an object much like a SETL 'tuple'), and the x appearing in f(x) or in an assignment f(x):=y must be an integer. This complicates the manipulation of attributes associated with arbitrary objects x (and attribute manipulation is basic to programming). To manipulate attributes of a non-integer object x (say string or a set) in these other languages, one must first find a way of associating an integer with x, and then must use this integer, instead of x itself, whenever the attributes of x need to be used or manipulated. This introduces a layer of artifice into programs, making them less direct, less readable, and more error- prone. This objection applies even to a language as elegant and powerful as APL, which only allows integers (and arrays of integers) to appear as indices. The only well-known languages which support something like the map operations of SETL are SNOBOL (through its TABLE feature) and some of the more advanced versions of LISP.

In deciding whether to use map operations like f{x} and f(x), or map assignments like f{x} := y or f(x) := y, it is important to realize that they are performed efficiently.

The internal representation of a map f (described in more detail in Sections 10.2, 10.4) makes it easy to locate all the pairs [x,y] of f which share a common first component x. This is done by using an exceptionally fast searching technique (known technically as 'hashing'). If the value of x is something relatively simple (like an integer or string) this makes it

possible to retrieve either of the values f(x) or f{x} in approximately a hundred millionths of a second (assuming that your program is running on a typical modern computer able to perform about a million addition operations per second). Note that the map operation f{x} is performed in a time which is essentially independent of the size of f. Similar remarks apply to the important Boolean set membership operation x IN s. See Section X for additional information on the way in which SETL objects are actually represented within the memory of a computer, and on the way that primitive SETL operatons, like the evaluation of f(x) or f{x}, are implemented.

2.7.3 Some remarks on Multi-Valued Maps

Set-valued maps can be handled (in SETL) in one of two nearly equivalent styles. Either style is acceptable, and neither has any overwhelming advantage, but they are different, and to avoid error it is important to distinguish clearly between them. These two possibilities are as follows:

(i) A set-valued map f can be represented as a single-valued map whose value f(x) is a set; but

(ii) The same map can be represented by a multivalued map g such that g{x}=f(x).

If f is available, then g can be produced by writing

(10) g:={[x,y]: s=f(x), y IN s}

Conversely, if g is available, then f can be produced by writing

(11) f:={[x,s]:s=g{x}}

(See Section 3.3.6 for an explanation of the 'map iterator' construct s = g{x} appearing in (11). Note also that if (10) is followed immediately by (11), then elements x such that f(x)={ } will drop out of the domain of f.)

A new pair [x,y] can be added to g simply by writing

```
g WITH:= [x,y]
```

(See Section 2.12.1 for an explanation of the 'assigning operator' WITH:= appearing here.)

The equivalent transformation of f must be written

```
f{x} WITH:= y;
```

which is a bit clumsier (see Sections 2.12.1 and 2.12.2).

To initialise g to a set of pairs defined by a condition C, one would normally write something like

```
g:={[x,y]: x IN s1, y IN s2|C}
```

The corresponding initialisation of f, namely

```
f:={[x,{y IN s2|C}]:x in s1};
```

is a bit clumsier.

These small technical differences sometimes lead one to prefer the 'g' representation of set_valued maps to the 'f' representation.

2.7.4 Two useful map operations

The 'inverse' of a map g is the map h such that [x,y] IN h if and only if [x,y] IN g. (If g is single-valued, this is equivalent to the condition that y=h(x) if and only if x=g(y)). We can produce h from g simply by writing

```
h:={[y,x]:[x,y] in g};
```

This important construction occurs frequently.

The 'product' or 'composite' of two maps g1, g2 is the map G such the [x,y] IN G if and only if there exists a z such that [x,z] IN g1 and [z,y] in g2. (If g1 and g2 are both single-valued , this is equivalent to G(x)=g2(g1(x)).) To produce G from g1 and g2, we can simply write

```
G:={[x,y]: z=g1{x}, y IN g2{z}};
```

or, in the single-valued case,

```
G:={[x,g2(z)]: z=g1(x)|g2(z)/=OM};
```

This 'map product' operation is also quite important. Note for example that if Fa maps each person x onto the father of x, and Mo maps each person y onto the mother of x, then the composite of Mo and Fa maps each person x onto x's paternal grandmother, while the composite of Fa by Mo maps each x onto x's maternal grandfather.

2.7.5 Multi-parameter maps

As noted above, maps f are used to associate attributes f(x) or sets f{x} of attributes with elements x. It is occasionally necessary to deal with attributes f(x1,..,xk) that depend in two or more objects x1...xk. For this purpose, the generalised map operations

```
(1a)    f{x1,...,xk}
(1b)    f(x1,...xk)
```

and the corresponding map asignments

```
(2a)    f{x1,...,xk}:=y
(2b)    f(x1,...,xk):=y
```

are provided. These simply abbreviate

```
(1a')     f{[x1,...,xk]}
(1b')     f([x1,...,xk])
```

and

```
(2a')     f{[x1,...,xk]}:=y
(2b')     f([x1,...xk]):=y
```

respectively. That is, a 'multiparameter' map f(x1,...,xk) of k parameters is regarded simply as a map whose domain consists of tuples of length k. Note that such a map cannot be used as a function ofany smaller number of parameters, since for j < k we will always have f{x1,...xj}={ } (except for j=1, where of course we have f{[x1,...,xk]}=f(x1,...,xk)).

All SETL's map constructs can be used with multi-parameter maps if they are regarded as one parameter maps whose domain elements are tuples. For example, if f is a k-parameter map, then the setformer

```
{y:z=f(y)}
```

will form the domain DF f by iterating over all the k-tuples y in DF. (See Section 3.3.6 for additional material concerning the 'map iterator' construct appearing here.)

2.8 Compound Operators

Binary operators like + or * are often used to sum or multiply all the components or members of a set or tuple, as in

```
t(1) + t(2) +...+ t(n)
```

To make it more convenient to form combinations of this kind, SETL allows any binary operator sign (including both built-in operators and user-defined binary operators introduced by OP declarations, see Section 4.7.2 below) to be followed immediately by a / (slash) mark. This introduces so called compound operators, such as +/ or */. Such operators can be used either in prefix or in infix position, i.e. either as

(1A) bop/t

or as

(1B) x bop/t

The t appearing in (1A) or (1B) must be either a set or tuple. The prefix form (1A) of the compound operator represents the result

(2A) e1 bop e2 bop ... bop en

obtained by combining all the elements or components of ej of t together using the underlying binary operator bop repeatedly. The infix form (1B) is similar, but also includes its first argument x in the result, i.e. forms

(2.b) x bop e1 bop e2 bop ...bop en,

where again the ej are all the elements or components of t. If t is null, the value of (1A) is OM and that of (1B) is x; if t has just one component or element e1 the (1.1 represents x bop e1, and (1A) simply represents e1, i.e. does not involve any application of -bop-.

The following are some typical uses of compound operators:

```
+/t     $ sum of all the elements of t, OM is t is null
0+/t    $ sum of all elements of t, 0 if t is null
MAX/s   $ maximum element in s, OM if s is null
0 +/[a(i)*b(i):t in [1..#a]]  $ dot product of a and b
*/[x IN t|x/=0]  $ product of all the nonzero components of t
```

As these last two examples illustrate, when a compound operator is used to combine an explicitly given sequence of terms, a tuple former should normally be used. If a set former is used then duplicate elements will only appear once, as in

```
0+/{x in t|x0}  $ duplicate elements not summed
```

Moreover, the SETL compiler recognises expressions which apply compound operators to tuple-formers and implements them efficiently, without actually building an unnecessary tuple. For example, the sum t/[2*t(i): i IN [1..#t]] is formed simply by iterating over t; no tuple is actually built.

The compound operator -bop/- can be formed with either built-in binary operators of SETL or with user-defined binary operators. For example, if, using the mechanisms described in Section 4.7.2, one introduces an infix operator COMP which forms the composite -f COMP g- of two maps, as defined by the formula

```
f COMP g = {[x,f(y)]:x IN DOMAIN g, y IN g{x}|f(y)/=OM},
```

then COMP/t will form the composite fn COMP... COMP f2 COMP f1 of a sequence [f1,f2,..,fn] of maps, and therefore COMP/[f:k in [1..n]] will form the 'nth power' of the map f, i.e the result of taking its composition with itself n-1 times.

2.9 Types and type-testing operators

The possible types of SETL values are Atom, Boolean, Integer, Real, String, Set, and Tuple. The built-in monadic primitive operator TYPE applies to any operator and produces its type, as a capitalised string. I.e., for any x TYPE x is either 'ATOM', 'BOOLEAN', 'INTEGER', 'REAL', 'STRING', 'SET', or 'TUPLE'. The language also provides a set of built in binary operators called IS_ATOM, IS_BOOLEAN, IS_INTEGER, IS_STRING, IS_SET, IS_TUPLE, each of which yields TRUE if applied to an object of the corresponding type, FALSE if applied to an object of any other type.

One additional monadic operator, IS_MAP, yields TRUE when applied to a set all of whose elements are pairs, FALSE otherwise.

The undefined value Om cannot be expected to have a type, and indeed the expression TYPE(OM) yields OM itself. In addition, any of the type predicates, such as IS_SET(OM) or IS_ATOM(OM), yields FALSE.

2.10 The ? Operator

In certain situations undefined (i.e. OM) results can be expected to appear and one will want to replace them by some other default values when they do appear. A typical situation of this kind is that in which one is counting the number of occurences of words in text: here it is natural to use

```
count(wd):=count(wd)+1 ;
```

to update a map -count- representing the number of times each word -wd- has been seen. But then, if -wd- has never been seen before, count(wd) will be OM, and we will want to replace OM by the more meaningful default 0. To do this we could write (using a syntax to be described more precisely in Ch. 3)

```
(1) count(wd):=IF count(wd)=OM THEN 0 ELSE count(wd) + 1 END ;
```

however, since constructs like this occur so frequently, an abbreviation x?y, which makes them easier to express, is provided. The definition of x?y is simply

```
IF (temp:=x)/=OM THEN temp ELSE y END,
```

where -temp- is a compiler-generated variable not otherwise accessible to the SETL user. Using this convenient operator, we can write (1) very conveniently as

```
count(wd):=count(wd)?0+1.
```

2.11 Exercises

Ex. 1 Write a program which calculates the set of all integers from 2 to 100 which are the product of exactly two primes.

Ex. 2 The Goldbach conjecture states that every even number greater than 2 can be written as the sum of two prime numbers. Write a 1-statement SETL program which verifies that this conjecture is true for the first 100 even numbers.

Ex. 3 Which of the following equations are valid for all tuples t1,t2,t3 and positive integers n,m?

```
(a)    t1+t2= t2+t1
(b)    t1+(t2+t3)= (t1+t2)+t3
(c)    #(n*t1)= n*#t1
(d)    n*(t1+t2)= n*t1+n*t2
(e)    (n+m)*t1= n*t1 + m*t1
(f)    (n*m)*t1= n*(m*t1)
```

If an equation is not always true, give an example showing a case in which it is false.

Ex. 4 Given a tuple t, write an expression which forms a tuple t1 in which every distinct component of t occurs exactly once. For example, if t is [1,2,1,2,3,3], t1 should be [1,2,3]. Also, write an expression which forms the set of all components of t which occur at least twice in t.

Ex. 5 Given a tuple t, write an expression which counts the number of non-OM components of t. Also, write an expresion that produces a tuple with the same components as t, but in reverse order.

Ex. 6 What are the values of the following Boolean expressions?

```
(a)    [1,2,[3,4]]=[1,2,3,4]
(b)    3 IN [1,2,[3,4]]
(c)    #[1,2,OM,3,OM]=4
(d)    [1,2,[3,4],OM]/=[1,2,3,4]
(e)    [1..4]=[1,2,3,4]
```

Ex. 7 The tuple t is [1,OM,2,OM,3]. Evaluate the following sequences:

```
(a)    t(1),t(2),t(3),t(4),t(5)
(b)    t(1..1),t(2..2),t(3..3),t(4..4),t(5..5)
(c)    t(1..),t(2..),t(3..),t(4..),t(5..)
```

Ex.8: write a tuple former that constructs the sequence of all prime numbers from 2 to 100, in ascending order.

Ex. 9 The tuple t is ['Tom', 'Dick', 'Harry', 'Sue', 'Lois']. Write a tuple-former whose components are those components of t which contain at least two vowels.

Ex. 10 Write a tuple assignment of the form t(i..j):=x which will convert

the tuple t=[1,2,3] to each of the following:

(a) [4,5,6,7] (b) [ ] (c) [1,3] (d) [1,OM,OM,3] (e) [1,4,10,3]

Ex. 11 Write a program that reads a tuple t of numbers and prints its three largest components in decreasing order.

Ex. 12 Changing as few of the elements of the set {[1,2], [3,4], [5], [ ]} as possible, produce a set s such that IS_MAP(s) evaluates to true.

Ex. 13 Given a tuple t of integers, write an expression which yields the index of the largest component of t.

Ex. 14 Assuming that s1 and s2 are non-null sets of integers, in what cases do the equations

```
+/(s1+s2)=+/s1 +/s2
```

and

```
*/(s1+s2)=*/s1 */s2
```

hold? What happens if s1 or s2 is null? How can we keep the null case from being exceptional?

Ex. 15. Write a definition of the sets DOMAIN f and RANGE f using set formers.

Ex. 16. The inverse INV f of a map f is the set of all pairs [y,x] for which [x,y] belongs to f. Express INV(INV f) in terms of f using a setformer.

Ex. 17. Given a map f, express the set s of all x for which f(x) is different from OM in terms of f. What is the relationship between s and DOMAIN f? In particular, when are s and DOMAIN f identical?

Ex. 18. Express the condition

```
[x,y] IN f
```

in terms of the image set f{x}.

Ex. 19. Let f denote the set

```
{[i,j]: i in [1..10], j in [1..10]| i>j}
```

What is the domain of f? What is the range of f? For what x is f(x) different from OM? What is f{5}? What is f(5)? What is the inverse map g (cf. Ex.16) of the map f?

Ex. 20. Answer question 19, but for the set f defined by the set former

```
{[i,i*i]: i in [-5..5]}
```

Ex. 21. Answer question 19, but for the set f defined by the set former

```
{[i,i*(i-1): i in [-5..5]}
```

Ex. 22. The map f has the set of strings 'Tom', 'Dick', 'Harry', 'Louis' as its domain; the map f1 has 'Sue', 'Mary', 'Helen', 'Martha' as its domain. Each of these maps sends every string element s of its domain into the length s of s. The maps F and F1 are the inverses of f and f1, respectively (see Ex.16). Answer question 19, but for the sets F and F1, the union set F + F1, and the intersection F*F1.

Ex. 23. Let f be the map

```
{[i,i*1]: i in [-2..2]}
```

(a) Write a series of map assignments of the form f(x):=y which will make f equal to the nullset { }. (b) Write a series of such assignments which make f single-valued by reducing its domain progressively. (d) Write a series of such assignments which make f single-valued without ever changing its range.

Ex. 24. The range of a map is the null set { }. What is the domain of the map? What is the map?

Ex. 25. The range of a map consists of the two elements {TRUE,FALSE} and its domain consists of the three elements {1,2,3}. (a) How many elements can the map itself contain? (b) How many such maps are there? (c) How many such single-valued maps are there? (c) How many such maps whose domain includes all three elements {1,2,3} are there? (d) How many such maps whose range includes both elements {TRUE,FALSE} are there? (e) Can you write SETL expressions which would evaluate the answers to all these questions?

Ex. 26. (a) The range of a map consists of the two elements {TRUE, FALSE}. How many elements can the map itself contain? (b) The domain of a map f consists of the three elements {1,2,3}. How many elements can the map itself contain? If we suppose that f is single-valued, how many elements can the map itself contain, and how many elements can its domain contain?

Ex. 27 A set s is a subset of every other set. What is s? A map f is a subset of every other map. What is f?

Ex. 28 Suppose that the variable s has a set value, the variable t has a tuple value, and the variables s1 and s2 have string values. Write expressions which produce the following quantities:

(a) A tuple whose components are the elements of s, arraged in some order.

(b) A set whose elements are the components of t, with duplicates eliminated.

(c) A tuple whose components are the successive characters of s1.

(d) Assuming that s1 and s2 have the same length, a map from each character of s1 to the corresponding character of

Ex. 29 Given two sets s1 and s2, express #(s1+s2) in terms of #s1, #s2, and #(s1*s2). If s2 INCS s1 is TRUE, express #(s1-s2) in terms of #s1 and #s2.

Ex. 30 Given two sets s1 and s2, express the number of single-valued maps f such that DOMAIN f=s1 and RANGE f=s2 in terms of s1 and s2.

Ex. 31 The map part of a set s is the collection of all elements of s which are ordered pairs. Write an expression whose value for any given s is the map part of s. (Make sure that your expression can be evaluated for any value of s, whether or not this value is a set; if s is not a set, your expression should have the value OM.)

Ex. 32 The single-valued part of a map s is the set of all pairs in s whose first component is unique. taking the same precaution noted in Exercise 30, write an expression whose value for any given s is the single-valued part of s. s2.

2.12 General form of the SETL assignment. The operators FROM, FROME, and FROME.

In preceeding sections, we have observed that some of the constructs which can appear in an expression, and which retrieve values or parts of values, can also appear on the left hand side of an assignment, allowing the corresponding values to be assigned or modified. For example, when it appears in an expression the expression f{x} retrieves the image set of x under the map f, but when it appears to the left of an assignment, as in

```
f{x} := e;
```

then the image set of x becomes e. Similarly, when the expression s(i..j) appears in an expression it yields a string or tuple slice, but when it appears to the left of an assignment, as in

```
s(i..j) := e;
```

is causes the value of this string or tuple slice to become e.

Constructs which can appear to the left of an assignment operator can also appear in expressions, and the relationship between left-hand and right-hand appearances (i.e., ordinary appearances within an expression) of any such construct always exhibits an important logical symmetry. Specifically, if, -lhs- denotes any construct which, like the constructs f{x} and s(i..j), can appear to the left of an assigment, then the effect of the assignment

```
lhs := e
```

is to assure that immediately subsequent evaluation of -lhs- (within an expression, i.e., in a 'right-hand' context) will yield the assigned value e.

The elementary constructs which are allowed to appear to the left of an assignment operator are the following:

(i) A variable identifier x. The assignment

```
x := e
```

modifies the value of x.

(ii) A tuple-former [x1,..,xk].

(Notice that the elipsis: ,..., stands for some unspecified number of other components of the tuple. This should not be confused with the SETL substring operation s(x..y)).

The assignment:

```
[x1,..,xk] := e
```

modifies the value of each of x1,..,xk. In such an assignment, any of the

xj can be replaced by the dummy symbol '-' (dash), in which case no assignment is performed for this particular xj. (This is the one exception to the general rule that any construct which can appear to the left of an assignment can also appear to its right.) As an example of this, note that the assignment

(1a) [x,-,y] := [1,2,3];

gives x the value 1 and y the value 3. Moreover, the assignment

(1B) [x,-,y] := [1,2,3,4];

has the same effect, since the fact that y occurs as the third component of the tuple on the left of (1B) means that the third component of the right-hand side of (1B) will be assigned to y. For the same reason, the assignment

(1C) [x,-,y,z,w] := [1,2,3,4];

gives x,y,z, and w the respective values 1,3,4, and OM.

(iii) A tuple, string, or map selection f(x). The assignment

f(i) := e;

modifies component i of f if f is a tuple, character i of f if f is a string, and the value f(i) if f is a map.

(iv) A multiparameter map selection f(x1,..,xk). This is equivalent to f([x1,...xk]), and the assignment

f(x1,...,xk) := e;

is equivalent to f([x1,...,xk]) := e.

(v) A multivalued selection f{x}. The assigment

f{x} := e ;

modifies the set f{x}.

(vi) A multivalued, multiparameter map selection f{x1,...,xk}. This is equivalent to f{[x1,...xk]}, and the corresponding assignment

f{x1,...,xk} := e ;

is equivalent to f{[x1,...,xk]} := e;

(viii) A string or tuple slice t(i..j) or t(i..). The effect of

t(i..j) := e or t(i..) := e

is to modify the portion t(i...j) of the string or tuple. Note that the value of the string or tuple expression e may have a length different from that of the subsection of t which e replaces, so these assignments can increase or decrease length of t. See Sections 2.3.3 and 2.5.3, also Table 2.1, for a discussion of marginal cases of these assignments, e.g. j=i-1, i= t+1. etc.

Simple expressions, of any of the types we have just listed, which can appear on the left of an assignment, can also be compounded to build up more complex 'assignment targets' that are also allowed to appear to the left of an assignment operator. For example, if f and g are maps, t is a tuple, and s is a string, then the assignment

```
(1A)  [[x,y],f(u),g{v},t(i),s(j..)] := e
```

is a legal assignment, whose effect is the same as that of the following sequence of assignments

```
(1B)  [temp1,temp2,temp3,temp4] := e;
      [x,y] := temp1;
      f(u) := temp2;
      g{v} := temp3;
      s(j..) := temp4;
```

Map and tuple component extraction operators can also be compounded, e.g. we are allowed to write h{u}(v)(i) if h is a map such that H1=h{u} is also a map for which H1(v) is a tuple whose i-th component can be extracted. The value x that h{u}(v)(i) produces is exactly that produced by the sequence

```
H1 := h{u};
H2 := H1(v);
x := H2(i); $ H1 and H2 are otherwise unused, compiler-generated
            $ variables
```

Compounds of this sort can also be used to the left of assignment operators, for example we can write

```
(2A)   h{u}(v)(i) := e;
```

This has exactly the same effect as the following sequence, into which the SETL compiler expands (2A):

```
(2B)   H1    := h{u};
       H2    := H1(v);
       H2(i) := e;
       H1(v) := H2;
       h{u}  := H1;
```

The general rules used to expand compound assignments can be stated as follows:

(i) An assignment of the form

(3A) [e1,...,ek] := x

is legal if, for each j between 1 and k, either ej is the sign '-' (dash), or if an assignment of the form

ej := y

would be legal. If it is legal, (3A) is expanded into the code sequence

(3B)
```
e1 := x(1);
...
ek := x(k);
```

but in (3B) every assignment corresponding to an ej of the form '-' is omitted.

(ii) An assigment of one of the forms

```
(4A)  e(i) := x;
(4B)  e(i1,..,ik) := x;
(4C)  e{y} := x;
(4D)  e{y1,..,yk} := x;
(4E)  e(i..j) := x;
(4F)  e(i..) := x;
```

is legal if and only if e is an expression, other than a tuple- former [z1,..,zk], which could appear to the left of an assignment operator, and if in addition the corresponding code sequence

```
(5A)  temp_var := e; temp_var(i) := x; e := temp_var;

(5B)  temp_var := e; temp_var(i1,..,ik) := x; e := temp_var;

(5C)  temp_var := e; temp_var{y} := x; e := temp_var;

(5D)  temp_var := e; temp_var{y1,..,yk} := x; e := temp_var;

(5E)  temp_var := e; temp_var(i..j) := x; e := temp_var;

(5F)  temp_var := e; temp_var(i..) := x; e := temp_var;
```

would be legal. (Here, -temp_var- is an otherwise unused, compiler-generated auxiliary variable). When an operation (4A-F) is legal, it is expanded into the corresponding assignment sequence (5A-F). Of course, the final assignment of each of these sequences may itself require expansion; if necessary, this is performed recursively, leading to expansions like those shown in (1B) and (2B) above.

2.12.1 'Assigning forms' of infix operators. Assignment expressions.

SETL allows abbreviation of any assignment of the form

(6) lhs := lhs OP e;

where OP designates any built-in (or user-defined, see Section 4.7.2) infix operator, and -lhs- designates any simple or compound expression which can legally appear to the left of an assignment operator. For example, we can abbreviate

```
i := i+1;
```

and

```
x:=x MAX y;
```

as

```
i +:= 1;
```

and

```
x MAX:= y;
```

respectively. One is always allowed to abbreviate (6) as

(7) lhs OP:=e;

2.12.2 Assignment expressions

Simple assignments x := y (and even more complex assignments such as f{u}(v) := y) can be used as expressions. The value of such an 'assignment expression' is simply its right-hand side y, but of course 'evaluation' of such an 'expression' always has a side effect, namely it modifies the value of the variable x.

Assignment expressions of this sort are frequently used to abbreviate sequences of assignments which initialise a collection of variables by giving the same value to all of them. For example, the assignment

```
x := y := z := w := 0;
```

which is equivalent to

```
x := (y := (z := (w := 0)));
```

gives all four variables x,y,z,w the value zero. Another common use of assignment expressions is to save the value of quantities that one needs to use just past the point at which they are first evaluated. The code fragment

```
(8) IF (x := f(u) + g(v)) IN s THEN f(u) := x;  ELSE...
```

illustrates this. Since the quantity f(u) + g(v) is needed immediately after the test in which it is first evaluated, the programmer may find it convenient to assign this quantity as the value of an auxiliary variable x, saving re-evaluation, and, equally important, abbreviating the program

source text. A related example, showing another common use of the assignment expression construct, is

```
(9)  IF (x := y+z) > 0 THEN
        positives WITH:=x;
     ELSE
        negatives WITH:=x;
     END IF;
```

Over-enthusiastic use of assignment expressions will lead to a crabbed programming style in which important operations flit by without sufficient syntactic emphasis. This will be bad if it deprives a program's reader of too much of the redundancy on which his understanding of the program depends. A good rule of thumb is to use an assignment expression only when the subsequent target variable of the expression occurs within a very few lines after the assigning expression being written.

2.12.2.1 Other positions in which assignment targets are allowed

A few of the other positions in which variables can occur resemble the left-hand sides of assignment operators, in that new values are assigned to variables appearing in these positions when the contexts containing them are evaluated. These 'assigning positions' are as follows:

(i) The position of x in an iterator

```
(FOR x IN s|... )...
```

is assigning, since the iterator will assign successive values to x. The same remark applies to the position of x in an existential quantifier

```
EXISTS x IN s|...
```

and in a universal quantifier

```
FORALL x IN s|...
```

Of course, the same remark applies to variables appearing in corresponding positions in multiple iterators, as in the case of the variables x,y, and z in

```
(FOR x IN s, y IN t, z IN [1..n]|...)
```

(ii) The position of x and i in a map, tuple, or string iterator

```
(FOR x=f(i)|..)
```

or in a multi-valued map iterator

```
(FOR x=f{i}|..)
```

is assigning. Of course, the corresponding positions in multiple iterators and in quantifiers are also assigning positions. (See Section 3.3.6 for aditional material concerning the 'map iterator' construct appearing here.)

(iii) Argument positions in function and procedure invocations corresponding to formal procedure or function parameters (see Chap. IV) that carry the read/write qualifier RW are also assigning positions (see Section 4.5).

Precisely the same expressions that can appear to the left of an assignment operator are allowed to appear in any other assigning position. Thus, for example, the construction

```
(FOR x+y IN s|...)
```

is illegal, since

```
x+y := e
```

would also be illegal; x+y is not a legal assignment target. On the other hand,

```
(10A)    (FOR [x,y] IN s|C)...

(10B)    (For f(x) IN s|C)...

(10C)    (FOR [[u,v],y] IN s|C)
```

are all legal, and have the same respective meanings as the code fragments

```
(11A)  (FOR temp_var IN s) [x,y] := temp_var; IF not C THEN QUIT; END;

(11B)  (For temp_var IN s)f(x) := temp_var; IF NOT C THEN QUIT; END;

(11C)  (For temp_var IN s)[[u,v],y] := temp_var; IF NOT C THEN QUIT; END;
```

into which the SETL compiler expands them. Much the same remark applies to quantifiers containing iterators in assigning positions, for example in

```
(12) ...EXISTS [x,y] IN s|C(x,y)...
```

The iteration implicit in the existential quantifier (12) will generate successive elements z of s and perform an implicit assignment [x,y] := z before the Boolean expression C(x,y) is evaluated.

As already noted, the position of i in

```
(13A)    (FOR x=f(i)|..)...
```

and in

```
(13B)    (FOR x=f{i}|..)...
```

also the positions of il,...,ik in

```
(13C)    (FOR x=f(il,...,ik)|...)...
```

and in

```
(13D)    (For x=f{il,...,ik}|...)...
```

are assigning. (See Section 3.3.6 for additional explanation of the 'map iterator appearing in (13A-D).).

Any expression which can appear to the left of an assignment operator can be substituted for the i in (13A) or (13B), or for any of i1 thru ik in (13C) or (13D). For example, we can write

(14) (FOR [x,y]=f([u,v])| C(x,y,u,v))...

In (14), the iterator will generate successive elements z of the domain of f and w of its range, and then perform implicit assignments [x,y] := w and [u,v] := z before the Boolean expression C(x,y,u,v) is evaluated. Note also that (13C) and (13D) are equivalent to

(15C) (FOR x=f([i1,...,ik])|...)...

and

(15D) (For x=f{[i1,...,ik]}|..)...

respectively.

2.12.3 The operators FROM, FROME, and FROMB

A useful, but somewhat nonstandard operator on sets s, and two similar operators on tuples t, have assignment-like side effects. These are

```
(16)    x FROM s;
```

and

```
(17A)   x FROME t;     $ take x 'from the end' of t
(17B)   x FROMB t:     $ take x 'from the beginning' of t
```

The effect of (16) is to select an arbitrary element of s, assign it to the variable x, and remove the selected element from s. Thus (16), like (17A) and (17B), has a somewhat unusual effect in that it modifies two variables. If s is null then x becomes OM and s remains null.

The form (16) can also be used as an expression; when used in this way, it yields the value assigned to x.

Similarly, the effect of (17A) is to remove the last (non-OM) component of t and assign it to the variable x. If t is null, then x becomes OM and t remains null. The effect of (17B) is to remove the first component of t and assign it to the variable x. If this first component is OM, then x is becomes OM, but t is reduced in length by 1 (its leading OM component is removed). If t is null, then x becomes OM and t remains null.

Like (16), the forms (17A) and (17B) can be used as expressions. When used in this way they both yield the value assigned to x.

Note that, if t has OM components immediately preceding its last non-OM component, then (17A) can decrease the length of t by more than 1. For example, the sequence

```
t := [1,2,OM,OM,3];
x FROME t; y FROME t;
print(x,y,#t);
```

will produce the output

```
3 2 [1].
```

The position of x in (16), (17A), (17B) is assigning. Any expression which could appear to the left of an assignment operator can also appear in this position. For example, we can write

```
[x,y] FROM s;
```

this is equivalent to the sequence

```
temp_var FROM s; [x,y] := temp_var;
```

2.13 Operator precedence rules

The table in this section shows the precedence rules which determine the order in which the operators in an expression are evaluated. If two operators share a common operand, then the one with the higher precedence is evaluated first. If both operators have the same precedence, then the left hand one is evaluated first (i.e. operators of a given precedence level are evaluated in a left associative manner.)

Parentheses can be used freely to emphasize or alter the order of operations as determined by this table.

Precedence	Operators
11	:= (on left side) assigning operators (on left side) FROM (both sides)
10	All unary operators except NOT and the IS_xx operators.
9	**
8	* / MOD DIV
7	+ -
6	User defined binary operators
5	= /= < <= > >= IN NOTIN SUBSET INCS
4	NOT and the IS_type operators
3	AND
2	OR
1	IMPL
0	:= (on right side) assigning operators (on right side)

The following examples of equivalent expressions with and without parentheses illustrate the operation of these rules:

```
a + b + c * d
```

is the same as

```
(a + b) + (c * d)
```

```
a + b +:= c DIV d
```

is the same as

```
a + (b +:= (c DIV d))
```

```
a + CEIL b := c
```

is the same as

```
a + (CEIL (b := c))
```

2.14 Exercises

Ex. 1 Given that t is a tuple, explain the meaning of ?/t.

Ex. 2 Write a setformer which will produce the set of all proper subsets of a set s, i.e. the set of all subsets s1 of s which are different from s.

Ex. 3 Express #pow(s) in terms of #s. Is there any set such that s=pow(s)? For what sets is #pow(s)=1? Is there any set that #s=#pow(s)? Is there any set such that #pow(s)=2?

Ex. 4 Given two sets s and t, their Cartesian product cprod(s,t) is {[x,y]:s IN s, y IN t}. Express #cprod(s,t) in terms of #s and #t. If cprod(s,t)={ }, what are s and t? Express #cprod(s,t) in terms of #cprod(t,s).

Ex. 5 It can be shown that two set expressions e1 and e2 involving any number of variables x1,..,xn and formed using only the set union, intersection, and difference operations are equal for all possible set values of the variables x1,...,xn if and only if they are equal whenever each of these variables has one of the two values { } and {1}. Therefore, we can check a set-theoretic identity lke x*y=y*x simply by evaluating

```
#{[x,y]:x IN {{ },{1}}, y IN {{ },{1}}|x*y/=y*x}
```

and observing that its value is zero. Moreover since x INCS y is equivalent to x*y=y, this same technique can be used to check inclusions of the form e1 INCS e2. Using this technique, verify that the following set-theoretic identities and inclusions are true for all possible set values of x,y, and z:

(a) (x*y)=(y*x)
(b) (x+y)=(y+x)
(c) ((x*y)*z)=(x*(y*z)), also ((x+y)+z)=(x+(y+z))
(d) ((x+y)-z)=((x-z)+(y-z))
(e) (x*x)=x, also (x+x)=x
(f) (x-x)={ }
(g) ((x+y)*z)=(x*z+y*z), also ((x*y)+z)=((x+z)*(y+z))
(h) (x+(y-x))=(x+y)
(i) (x-(y+z))=((x-y)*(x-z))
(j) (x*{ })={ }, also(x+{ })=x

If f is a map and s is a set, then the image set of s under f, sometimes written f[s], is by definition the set {y:[x,y] IN f|x IN s}. The inverse image of s under f, sometimes written f_inv[s], is by definition the set {x:[x,y] IN f | y IN s}. These notations will be used in the next group of exercises.

Ex. 6 Express f[s] in terms of the sets f{x}, using a compound operator. What is f[DOMAIN f]? What is f-inv[RANGE f]?

Ex. 7 In how many ways can two pairs of parentheses be inserted into the expression

```
1 + 2 - 3 * 4 DIV 5
```

to produce a legal expression? Take twenty of these expressions and write their values. Do the same for

```
1 + 2 -- 3 * 4 DIV 5.
```

Ex. 8 Determine the type of the value of x in each of the following code fragments, assuming that the code shown executes without causing any error.

(a)
```
x := z+1;
```

(b)
```
x := z+'1';
```

(c)
```
x := z-{1};
```

(d)
```
x := z--[1];
```

(e)
```
read(x);
IF x>0 THEN print(x); END;
```

(f)
```
x := ARB s;
(FORALL y IN s|y>0)print(y); END;
```

(g)
```
IF EXISTS x IN s |#x(i...j)<j-i THEN print(x); END;
```

Ex. 9 Execute the programs

```
[A,A,A]:=[1,2,3]; print(A);
```

and

```
[A,B,A,B] := [1,2,3,4]; print(A,B);
```

What result do you expect? What is going on?

Ex. 10 Write expressions which will find the following positions in a string s:

(a) The position of the first occurence of the letter 'a' (b) The position of the second occurence of the letter 'a' (c) The position of the n-th occurence of the letter 'a' (d) The position of the last occurence of 'a' that is preceded by no more than five occurences of 'e'.

If the desired occurences do not exist, your expression should return the value OM.

Ex. 11 Write an expression which, given a tuple t of integers, forms the tuple t2 of all 'partial sums' of the components of t. That is, the j-th component of t2 should be the sum of components 1 thru j of t.

Ex. 12 A tuple t of tuples, all of the same length n, can be regarded as an m x n rectangular array of items. Write a program which rearranges this array by turning it 90 degrees, so that it becomes an n x m rectangular

array of items, represented by a tuple t2 of tuples all of length m. If this operation is repeated twice, what happens?

2.15 OMS and Errors

When an illegal operation or an operation having an undefined result is evaluated during the running of a SETL program, one of two possible things will happen. Errors classified (somewhat arbitrarily) as 'severe' will cause execution to terminate. In this case, a brief error indication will be placed at the end of the program's output file. Moreover, if the terminal dump option has been switched on (section 8.5.1.4 below explains how this can be done), a terminal dump will be written to the dump file specified; valuable hints concerning the cause of error can then be gleaned by examining this dump.

The following errors terminate execution:

(i) Type errors, e.g. an attempt to evaluate

1+{0}, 1.0+2, [0]+{1}, '1'+2. s{y} where s is a string or tuple etc.

(ii) Illegal use of OM, e.g. attempts to evaluate

{OM}, f(OM), OM IN s, s WITH OM, OM WITH x, etc.

(iii) String or tuple parameters which are grossly out of bounds, e.g. attempts to evaluate

s(0) or s(-1),

where s is a string or tuple.

(iv) Illegal file operations, e.g. attempts to manipulate files which have not been opened.

(v) Floating point operations which overflow out of the range of a particular SETL operation, and also conversions of very large integers to floating point form.

'Mildly erroneous', deliberately intended, operations whose result is undefined will return the undefined value OM. These include

(a) selection of an element from an empty set or tuple, as in

x FROM { }, x FROM [ ], x FROME [ ], or ARB { }

(b) evaluation of a map at a point at which it is undefined or multiple valued, as in f(0) or f(1) where f is

{[1,1], [1,2]};

also evaluation of an undefined component of a tuple. Since in these cases execution is not immedately terminated, it is possible to test for an OM result in this case, giving greater semantic flexibility. Some typical constructs exploiting this flexibility are:

```
IF (x FROM s)/=OM THEN...  $ test a set for nullity and extract
```

```
                                $ an element if not null

    IF f(x)/=OM THEN..          $ see if the map f is uniquely defined at x
```

On the other hand, since the legal uses of OM are severly restricted, unexpected OM values are likely to force error termination soon after they appear. Consequently, errors of this sort can generally be tracked down rather quickly.

CHAPTER 3

CONTROL STRUCTURES

Chapter 3. Control structures

Execution of a SETL program proceeds sequentially , one statement being executed after the other. In the simplest case, the order of execution is simply the order in which the statements are written in the program. For example, consider:

```
a := 1;
print('Initially, a = ', a);
a := a + 1;
print('Finally, a = ',a);
```

In this example, the variable a is assigned the value 1; then the first message is printed; a is then assigned the value 2; and finally the second mesaage is printed.

Only the simplest computations can be carried out by such straight-line programs. In order to perform more complex calculations, we need to be able to describe conditional computations, i.e. computations that are to be executed only when certain conditions are met, and we also need to program repeated computations, i.e. computations to be executed a number of times, (100 times, or for all elements in a set, or until a certain calculation converges, or as long as a certain value has not been reached, etc).

The order in which these more complex computations are to be executed is specified in the program text by means of language constructs commonly called control structures. In this chapter we will examine the most important control structures of the SETL language, namely: the IF statement, CASE statement, LOOP statement, and GOTO statement. The IF, CASE, GOTO and some variant of the LOOP constructs are commonly found in most modern programming languages, and are regarded as the basic tools of 'structured programming'. The LOOP construct in SETL is a bit richer than the loop constructs provided by most other languages, and some of its features are specially tailored for the objects that characterize SETL, namely sets, tuples and maps.

Chapter Table of Contents

3.1 The IF statement.

The IF statement is used to route program execution along one of several alternate paths, chosen according to some stated condition. An example is

```
IF balance > 0 THEN
    print('Your line of credit is: ', balance);
ELSE
    print('you are overdrawn by: ', -balance);
END IF;
print('Do you want additional information (y/n)?');
```

Here, the condition (i.e. whether the value of -balance- is positive or negative) determines which of two messages is printed. If the condition being tested is TRUE (i.e. the balance is positive) the statement following the keyword THEN is executed; if the condition is FALSE, the statement following the keyword ELSE is executed instead.

After execution of the statements in either branch of the IF statement, program execution continues from the first statement following the end of the IF. In the example above, after execution of one of the branches of the IF, the query - Do you want additional information(y/n)?-will be printed.

Any number of statements can appear in either branch of an IF statement. For example, we can write:

```
IF line >= 50 then
    page := page + 1;
```

```
        line := 1;
    ELSE
        line := line + 1;
    END IF;
```

In this case, if the condition - line >=50 - is true, then the assignments to page and line are performed; otherwise, -line- is incremented.

The syntax of the form of the IF statement shown above is:

```
IF condition THEN
    group of statements
ELSE

   group of statements
END optional tokens;
```

The construct condition denotes any boolean expression, (See Section 2.xxx) i.e. any expression which yields either TRUE or FALSE. The group of statements in each branch of the IF designates any sequence of executable statements, which can be assignments, control statements such as other IFs, loops, etc.

The end of the IF construct is indicated by the keyword END, followed optionally by the keyword IF, and by up to 5 of the tokens that follow the opening IF. This convention is particularly useful for clarifying the range of statements governed by IF statements nested within other IF's, and is used for other nested control structures as well. The following example illustrates the use of nested IF statements, and displays the convention we have just described for indicating the end of an IF.

```
IF a /= 0 then

    IF b**2 > 4.0*a*c  THEN
        discr := sqrt(b**2 - 4.0*a*c);
        print('rl = ', (-b + discr) / 2.0*a );
        print('r2 = ', (-b - discr) / 2.0*a );
    ELSE
        print('Complex roots');
        re_part := -b/2.0* a;
        im_part := sqrt(4.0*a*c - b**2) / 2.0*a;
        print('rl = ', re_part, '+i', im_part);
        print('rl = ', re_part, '-i', im_part);
    END IF  b**2;

ELSE
    IF b /= 0 THEN
        print('Single root: ', -c/b);
        print('degenerate equation: a = b = 0');
    END IF b /= 0;
END IF  a /= 0;
```

3.1.2 Omitting the ELSE branch of an IF statement.

Sometimes we want to perform a series of actions when a certain condition is

met, but to do nothing if it isn't. In this case it is possible to omit the ELSE branch of an IF statement, as illustrated in the following simple example:

```
IF token NOTIN keywords THEN
    print('Unrecognized operator: ', token);
END IF;
```

If the condition is true, the statement(s) following the THEN are executed; if the condition is false, the IF statement does nothing.

3.1.3 The null statement.

For reasons of readability, it is often advisable to indicate both branches of an IF statement, even if one of them is to do nothing. A -do nothing- statement is provided for this purpose. It is written thus:

```
        pass;
```

and causes no computation at all. This allows us to write the previous example as follows:

```
IF token notin keywords THEN
    print('Unrecognized operator: ', token);
ELSE pass;
END IF;
```

This can also be expressed as:

```
IF token in keywords THEN
    pass;
ELSE
    print('Unrecognized operator:', token);
END IF;
```

3.1.4 Multiple alternatives in an IF statement.

We often encounter the following programming situation: when the condition of an IF statement is false, we immediately perform another test to choose among another pair of alternatives, and so on. This can be expressed by means of nested IFs, but can be more clearly stated by 'continuing' the IF statement by means of a special construct to designate subsequent alternatives. In SETL, this is done using ELSEIFs, as shown in the following example:

```
IF month = 'February' THEN
    IF year mod 4 = 0 and year mod 200 /= 0 THEN
        days := 29;
    ELSE
        days := 28;
    END IF year;
ELSEIF month in {'september','April','June','November'} THEN
     days := 30;
ELSE
     days := 31;
END IF;
```

Here, three alternatives are being examined: whether month is February, or is one of the 30-day months, or is one of the remaining months. Any number of alternatives can appear in this more general IF construct, whose syntax is:

```
IF condition THEN
   group of statements
ELSEIF condition THEN
   group of statements
ELSEIF ...
    ...
ELSE
   group of statements
END optional tokens
```

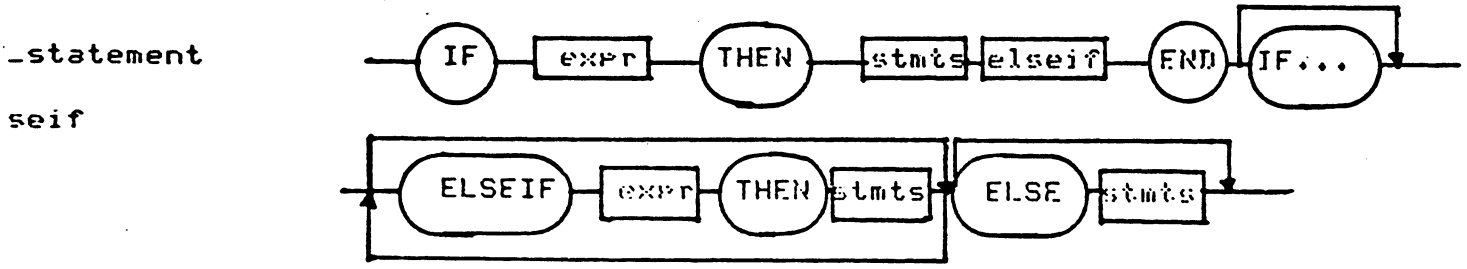


Fig 3.1: IF_Statement syntax diagrams

Note the important syntactic point:

- ELSEIF is a single word, and it indicates an alternate test within the current IF statement.

- ELSE IF, on the other hand, indicates that within the ELSE branch of the current IF statement, a nested IF statement is present, which will need its own closing END. Be warned: if you use ELSE IF when ELSEIF should be used, syntax errors, namely 'missing END' messages, will result.

3.1.5 An important note on indentation and programming style.

The physical layout of a SETL program on a printed page (or the screen) is of no concern to the SETL compiler. As long as the syntax of the language is obeyed, the user is free to write successive SETL statements with bizarrely varying indentation, to place several statements on the same line of text, etc. For the human reader, on the other hand, a good choice of program layout can make all the difference between clarity and hopeless muddle. This is particularly true when a program needs to to be read and understood by several programmers. Proper indentation should reflect program structure in such as way as to serve as an additional implicit documentation on the intent of a program. In this connection, the following maxim should be kept in mind: PROGRAMMING IS A SOCIAL ACTIVITY. If the programs you write are of any interest, there is a high likelihood that somebody else will want to examine them, so as to extend, modify or simply to understand their workings. (Often enough, this somebody else may be yourself, going back to a program written months before, trying to recapture the thought processes that led you to various programming decisions). In

other words, a program must be seen as a tool for communication, not only from programmer to computer, but also among programmers. From this perspective, it is easy to see that good indentation and program layout, helpful choice of variable names, and ample and carefully considered documentation, are the hallmarks which distinguish the professional programmer's work from that of the amateur.

In the case of IF statements, it is natural to regard the group of statements in each branch of the IF as subordinate to the the condition which introduces them. This is clearly reflected in the text if we INDENT the statements in each branch, with respect to the IF and ELSE keywords, as was done in the examples above. An additional rule to follow is to place the ELSE in a line by itself, unless the corresponding branch reduces to a single short statement (for example: pass;). The examples in this text follow these rules, as well as other ones which we will mention in connection with other control structures. As is usually the case for rules of style, these should only be regarded as guidelines and suggestions, to be tempered by individual taste. We cannot emphasize enough, however, the need for SOME consistent choice for indentation and paragraphing in the preparation of programs.

3.1.6 The IF expression.

An IF statement often is used to assign one of several values to a given variable. For example, one may write:

```
IF a > b then maxab := a;  ELSE maxab := b;  END IF;
```

When this is all that is wanted, the IF expression (also called conditional expression) provides a clearer way of achieving the same intent. An IF expression is an expression whose syntax is similar to that of the IF statement, and which denotes a value which depends on the outcome of a test (or tests). The general syntax of an IF expression is:

(1) IF test1 THEN expr1 ELSEIF test2 THEN expr2....ELSE exprn END

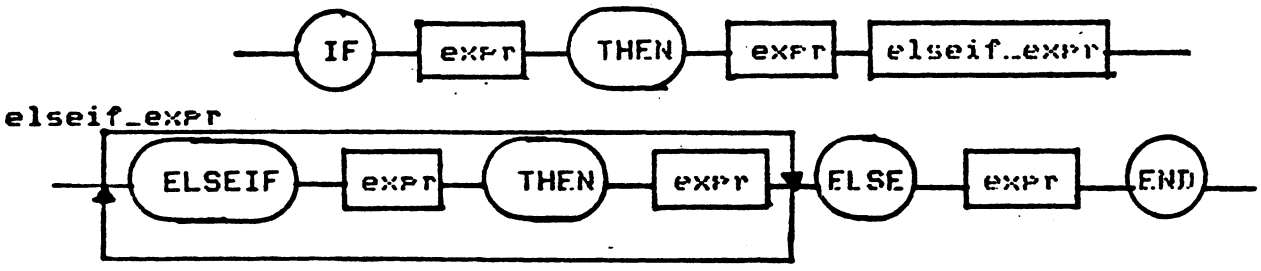


Fig 3.2: Syntax of the IF expression.

This construct may be used in any position where an expression of any other kind would be acceptable. For example the IF statement (1) can be written as:

```
(2)  maxab := IF a > b THEN a ELSE b END;
```

The following are also valid examples of IF expressions:

```
PRINT ( IF filler = '' THEN '***' ELSE filler + '*' END);
PRINT ((IF filler = '' THEN '**' ELSE filler END) + '*' );
distance := distance + (IF edge = OM THEN 0
                                    ELSE length(edge) END);
```

The following syntactic details of the IF-expression should be noted:

a)In an IF expression, an ELSE part must always be present (to insure that the expression has a value in all cases).

b) the terminator of an IF expression must be a simple END, not END IF or END IF with extra tokens. c) There is no semicolon preceding the keywords ELSEIF and ELSE in an IF-expression. This is because these keywords are preceded by expressions. In contrast, these same keywords are preceded by semicolons in an IF-statement, because in that case a semicolon terminates the statement previous to the keyword.

IF expressions can be nested, as the following rewriting of our 'days in the month' example shows:

```
days := IF month = 'February' THEN

(IF year mod 4 = 0 and year mod 200 /= 0 THEN 29

ELSE 28 END)

ELSEIF month in {'September','April','June','November'}

THEN 30

ELSE 31 END;
```

3.2 The CASE statement.

The CASE statement is a generalization of the IF statement. Whereas the IF statement controls the flow of execution of a program by choosing among two alternatives, the CASE statement allows us to choose among any number of alternative paths of execution. The CASE statement is available in two forms. Of these, the first and most general is:

```
CASE OF
(test1): block1
(test2): block2
(test3): block3
..
..
(testn): blockn
ELSE blocke
END;                                $ Or END CASE;
```

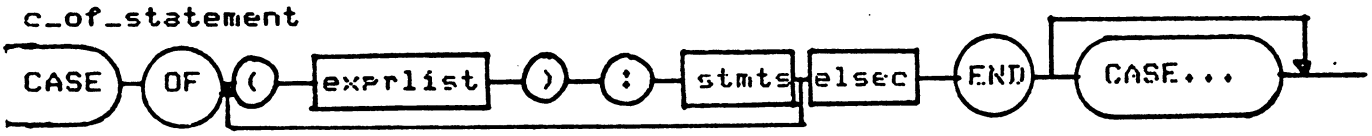


Fig 3.3: CASE-OF statement syntax diagrams

Each of block1, block2.. and blocke must be a sequence of one or more statements. Each of the expressions test1,test2.. must be a boolean expression. Execution of this form of the CASE statement proceeds as follows:

a) The expressions test1, test2..are evaluated. If one of them, say testi, yields TRUE, then the corresponding block. i.e. blocki, is executed, and then execution proceeds to the first statement that follows the CASE statement. If several of the expressions test1,test2.. evaluate to TRUE, then any one of them is chosen and the corresponding block executed. The CASE statement thus differs from a similar sequence of IF and ELSEIF statements, where the tests are made in sequence. b) If none of the tests evaluates to TRUE, then blocke, which follows the ELSE clause of the statement, is executed. This ELSE clause is optional. If the ELSE clause is absent, and none of the tests in the CASE statement evaluates to TRUE, the CASE statement is simply bypassed, and execution continues with the first statement that follows it.

It is possible to attach more than one test to a given branch of the CASE by writing:

```
(test1,test2...testJ): blockn
```

In this case, blockn is a candidate for being executed if any one of the tests test1,test2.. yields TRUE.

As a first example of the use of a CASE statement, the following SETL fragment calculates the volume of various geometric figures:

```
CASE OF
(figure = 'CUBE'):
    volume:= side ** 3;
(figure = 'SPHERE'):
    volume:= (4/3) * PI * radius ** 3;
(figure = 'CYLINDER'):
    volume:= PI * radius ** 2 * height;
ELSE
    print('Sorry, I don''t recognize this figure');
    volume:= 0;
END CASE;
```

As this example shows, it is quite common for the tests in a CASE statement simply to test a particular variable or expression for equality with a series of constants. The following second form of the CASE statement simplifies the writing of CASE statement of this kind:

```
CASE expr OF
(constant1): block1
(constant2): block2
.
.
(constantn): blockn
ELSE blocke
END;                        $ OR more generally END CASE tokens;
```

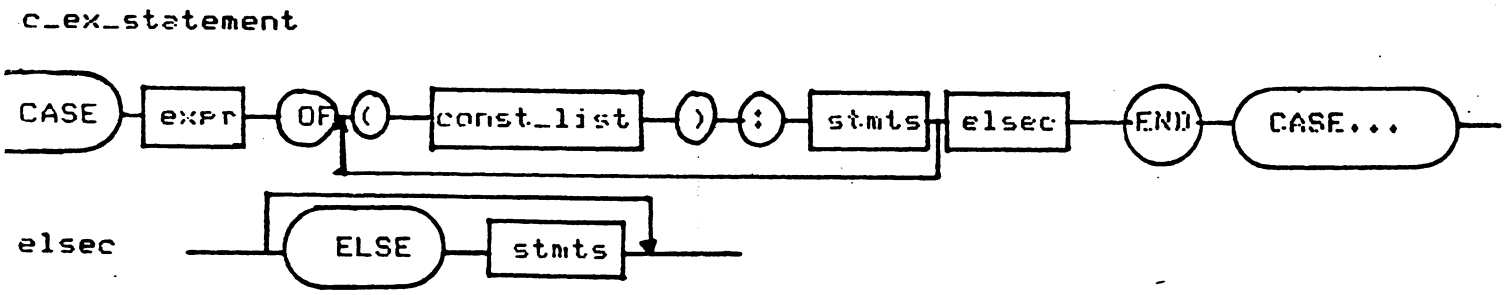


Fig 3.4: CASE-EXPN-OF statement syntax diagram

The expression in the header is evaluated (once) to give a test value. If the evaluation yields one of the constants prefixed to a branch of the case, say constanti, then the associated block blocki is executed. The ELSE block is executed if the value of -expr- does not appear as the prefix of any branch of the CASE statement. The ELSE block can be omitted if no action is to be taken when this happens. As in the first CASE statement form, multiple tests can be attached to one branch by writing:

```
(constant1,constant2..constantn): block
```

If this is done, the block will be executed if the value of the expression in the CASE header equals any of the values constantl..constantn.

3.2.1 The CASE expression

One will sometimes want to use a CASE construct simply to assign one of several alternative values to a variable. This can be done with a CASE statement, for example:

```
     CASE day of
(Sunday): discount:= 0.6;
(Saturday): discount:= 0.4;
(Monday,Tuesday,Wednesday,Thursday,Friday):
            discount:= 0.0;
END CASE;
```

In this example, the purpose of the CASE Statement is simply to assign an appropriate value to the the variable -discount-. The CASE expression allows this kind of thing to be written in a way that makes their purpose clearer. A CASE expression can appear wherever an expression can appear. Its syntax can be that described by either of the following syntax diagrams:

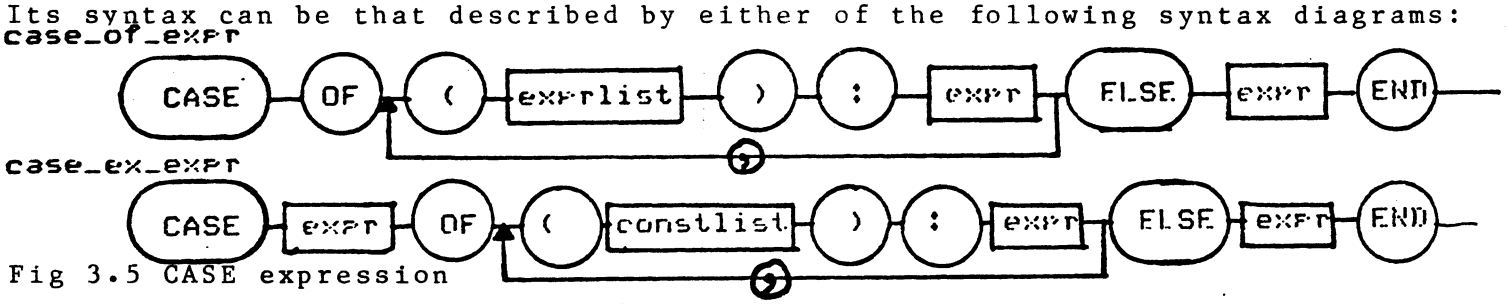


Fig 3.5 CASE expression

Evaluation of a CASE expression closely resembles that of the CASE statement. The execution of a CASE expression of the form (1) proceeds as follows: a) The expression following the CASE keyword is evaluated, yielding some value V. b) If V equals the value of one the constants that mark each branch of the CASE expression, then the value of the expression tagged by that constant is the value of the CASE expression. c) If none of the constants equal V, then the value of the expression that follows the keyword ELSE is the value of the CASE expression. Using this construct, the preceding example can be rewritten as follows:

```
discount := CASE day of
                (Sunday):   0.6,
                (Saturday): 0.4
            ELSE 0.0
            END;
```

Note that a comma is used to separate successive alternatives of the CASE expression, and that no comma appears before the ELSE keyword.

The second form of the CASE statement has no ruling expression, and each case is marked by a list of expressions,each of which must yield a boolean value. The value of CASE expression is the value of the expression tagged by a value of TRUE.

3.3 Loops.

There are several ways of constructing programs out of elementary statements. In Sec.3.1 we examined one of them: the IF statement, also called the alternation or conditional statement. We now turn our attention to iteration, or looping.

Almost every program involves some iteration. Whenever we need to deal with aggregates of data (all the books in a catalog, all the students in a class, all the prime numbers less than 1000, etc.) we are apt to specify some computation that is to be performed repeatedly. For example, we may want to do the following:

a) List all the members of a set (For example, all the students registered in a given course).
b) Modify each component of a tuple. (For example, discount all entries in a price list by 10 %).
c) Modify selected members of a tuple, for example raise the tax charged to every Texas resident appearing in a tuple by 6%, while leaving unchanged the taxes payed by residents of other states.

We may even want to perform an action repeatedly when no data aggregates are involved. For example:

d) Perform a series of actions a stated number of times. (E.g., print the string -*-*-*-*-*-*- 10 times).
e) Perform a series of actions as long a a certain condition is true. (E.g. to estimate the logarithm (base 2) of a number, we can divide it repeatedly by two as long as the result is greater than one, and count the number of times the division is performed).
f) Perform a series of actions until some condition is met. E.g. read input data until an End-of-file is detected.

The first three types of looping are expressed in SETL by using set and tuple iterators. Iterations of type d) are expressed using numeric iterators. Types e) and f) correspond to WHILE and UNTIL loops respectively. As we will see subsequently, SETL allows us to combine all these ways of expressing a repeated calculation into a very general Loop construct.

We now start our review of these various loop constructs, beginning with the simplest and most 'natural' ones: the set and tuple iterators. We have already encountered various iterator forms when we discussed tuple and set formers. We will now examine them in greater detail.

3.3.1 Set iterators.

The set iterator is used to specify that a certain calculation is to be performed for each of the elements in a given set. In its simplest form, it reads as follows:

```
LOOP for x in S DO                    (1)
  list of statements
```

```
end optional token;
```

The keywords LOOP and DO can be replaced by left and right parentheses, respectively, and we will often write our iteration loops using this shorter form:

```
(FOR x IN S)   list of statements     END     tokens;
```

The meaning of (1) is as follows:

a) Obtain the elements of set S in succession.
b) Perform the list of statements once for each element of S.
c) During successive iterations of the loop, assign the value of successive elements of S to variable x.

For example, suppose that S is the set:

```
 {'Springfield', 'Albany', 'Sacramento',  'Boston'}
```

Then the loop:

```
(FOR city IN S)
   print(city, ' is a state capital.');
END;
```

will produce the following output:

```
Springfield is a state capital.
Albany is a state capital.
Sacramento is a state capital.
Boston is a state capital.
```

The variable x in the construct 'x in S' is called the bound variable of the iterator, or simply the iteration variable, or loop variable. As you can see from the example above, its name is arbitrary. We chose to call it 'city' in this case but we could have called it 'c', or 'capital_city', or whatever, i.e. exactly the same output would have been obtained with the loop:

```
(FOR c in S) print(c, 'is a state capital.'); END;
```

Each time the list of statements (also called the loop body) is executed, the bound variable is assigned the value of another element of S. The loop body is executed exactly as many times as there are elements in S. When all elements of S have been dealt with, the program moves on to execute the statements that follow the end of the loop.

Consider the following example:

```
Fib13:= {1,1,2,3,5,8,13,21,34,55,89,144,233};
count:= 0;
(FOR N IN Fib13)
   IF N MOD 3 = 0 THEN
        PRINT(N, ' is a multiple of 3');
        count:= count + 1;
   END IF;
END;
PRINT('There are ', count, ' multiples of 3 in Fib13');
```

The purpose of this short code fragment is to list the multiples of 3 that appear in the set Fibl3 (which happens to be the set of the first thirteen so called 'Fibonacci' numbers). Each element of Fibl3 is tested for divisibility by 3, and printed if the test succeeds. A count is kept of the multiples of 3 that we encounter, and this count is printed at the end. The output of this program is:

```
3 is a multiple of 3
144 is a multiple of 3
21 is a multiple of 3
There are 3 multiples of 3 in Fibl3
```

You may be surprised by the order in which the numbers 3, 144 and 21 appear in the output. Why are they not listed in the same order as in the set Fibl3 ? The reason is of course that sets have no particular ordering, and when we iterate over a set, we don't know in what the order its various elements will be obtained. All we know is that we will obtain all of them, in some order, and that is all that matters. (When order matters, we must use tuples instead of sets. More about this below).

The bound variable that appears in a set iterator receives its values from successive elements of the set over which we iterate. When the iteration is complete, that is to say when all elements of the set have been assigned to the loop variable, the loop variable gets the value OM. The following loop:

```
(FOR number IN {1,3,10} + {15,30})
   print('number is: ', number);
END;

print;
print('Now number is: ', number);
```

produces the output:

```
number is: 3
number is: 1
number is: 15
number is: 10
number is: 30

Now number is *
```

Note two things in this example:

a) We can iterate over any <u>expression</u> whose value is a set. (I.e. the expression does not have to be a simple variable).

b) OM, the undefined value, is printed as an asterisk ('*').

c) The command

```
print;
```

by itself, i.e. without any arguments, prints a blank line.

The reason for calling the loop variable a BOUND variable should be clear:

the values taken by the loop variable are controlled by the iteration mechanism; the programmer cannot modify this sequence of values by means of assignment statements within the body of the loop.

Assignments to the loop variable within the body of the loop are not explicitly forbidden by SETL, but should be avoided on stylistic grounds. Note that such assigments have no disastrous consequences; they simply do not affect the course of the iteration. Consider the following fragment:

```
(FOR x IN {1,2,3})
   print('x = ', x);
   x:= x + 1;
   print('after increment, x = ', x);
END FOR;
```

The output of this fragment will be something like:

```
x = 3
after increment, x = 4
x = 2
after increment, x = 3
x = 1
after increment, x = 2
```

Note that the values received by the loop variable during this iteration are 3,2,1, regardless of the extra assignments to x in the loop body.

3.3.1.2 Conditional set iterators.

Consider the following problem: the holdings of a library are described by means of a set CATALOG and a series of maps: AUTHOR, SUBJECT, and so on. We want to list those books in the catalog whose subject is calculus. This can be achieved by means of the program fragment:

```
(FOR book IN CATALOG)
    IF SUBJECT(book) = 'calculus' THEN
        print(book);
    END IF;
END FOR;
```

The same effect is achieved by the following code:

```
(FOR book IN CATALOG | SUBJECT(book) = 'calculus')
    print(book);
 END FOR;
```

The vertical bar: '|', already introduced in sec. 2.xxx, is read 'such that', so that the last iterator can be expressed in English as follows: 'iterate over the elements of CATALOG which are such that their subject is "calculus"'. In other words, the 'such that' construct appearing in a conditional iterator allows us to specify an iteration over a specified SUBSET of a given set.

The general form of the conditional iterator is the following:

(FOR name IN set expression | boolean condition)
    list of statements
END optional tokens;

imple_iterator

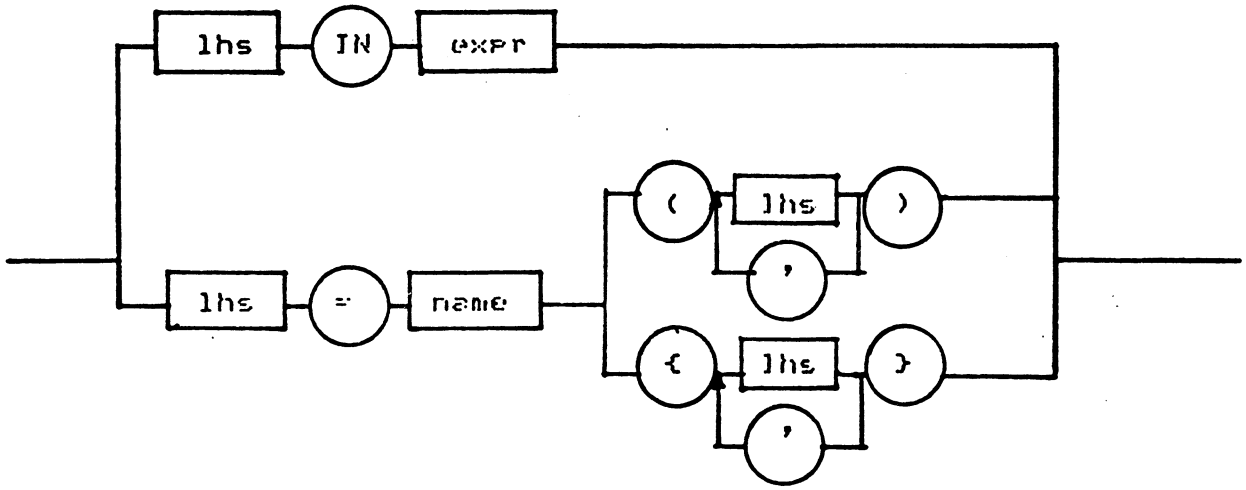


Fig 3.6: Simple iterator syntax diagrams

In this construct, boolean condition designates any predicate expression i.e. any expression that yields either TRUE or FALSE as its value. The meaning of this construct can be stated as follows:

a) Iterate over the elements of set expression, and assign the successive values of these elements to name.

b) After each of these assignments, evaluate the boolean condition. If the condition yields TRUE, perform the list of statements. Otherwise, skip directly to the next value of set expression.

Typically the iteration variable will appear in the boolean condition. This is shown in our previous example.

However it is possible, though inelegant, to write a conditional iteration whose boolean condition does not depend on the iteration variable. For example:

(FOR x IN S | TRUE)

is equivalent to:

(FOR x IN S)

because the boolean condition is -true- for all elements of S.

The following iteration is less artificial than the preceeding example:

```
(FOR x IN S | flag)                (2)
```

where -flag- is some boolean variable. It selects the elements of S according to the current setting of -flag-. This variable may be set elsewhere in the program, perhaps in the body of the iteration loop. However, the intent of (2) is expressed more clearly by the equivalent code:

```
(FOR x IN S)
    IF flag THEN ....
```

which should be preferred to (2) on stylistic grounds.

3.3.2 Tuple iterators, first form.

Iterations over tuples can be described in exactly the same manner as iteration over sets. That is, they can be given the form:

```
(FOR name IN expression | boolean condition)
    list of statements
END optional tokens;
```

If expression is a set expression, the loop is a set iteration. If expression yields a tuple, it is a tuple iteration. One significant difference between set and tuple iterators is that for the latter we know the order in which the components of the tuple will be examined by the iteration. Namely, they are produced in order of increasing index. For example

```
width := [1,3,5,7,9,2,2];
(FOR w IN width) print( w * '*' ); END;
```

always produces the ouput:

```
*
***
*****
*******
*********
**
**
```

In this example, the iteration variable w takes on the values of the components of the tuple -width-, exactly in the order in which they occur: first 1,3,5,7,9, and finally 2,2. (Question: what would the picture look like if we had defined width as {1,3,5,7,9,2,2} ?)

If a boolean condition is present, the tuple iterator obeys the same rule as the set iterator: the body of the loop is executed only for those tuple components for which the condition yields true.

3.3.3 String iterators, first form.

An iteration over a character string is specified in exactly the same manner as an iteration over a tuple. The following example illustrates this.

```
no_vowels := '';

(FOR c IN 'antidisestablishmentarianism' | c NOTIN 'aeiou')

    no_vowels +:= c;
END FOR c;
print(no_vowels);
```

The output of this loop is the string: 'ntdsstblshmntrnsm'

The action of a string iterator is very similar to that of a tuple iterator: successive components (in this case characters) are assigned to the loop variable, and the body of the loop is executed for those values of the loop variable that satisfy the stated boolean condition. The characters are iterated over in the order in which they appear in the string, from left to right.

3.3.4 Numerical Iterators.

An iterative computation is often expressed as follows: 'Repeat the following calculation N times '. Here the iterative process does not depend on a data aggregate, such as a set or a tuple, but rather depends on an integer, namely the value of N. Such iterations are in fact the type of iterative construct most commonly supported by other programming languages. In SETL, this type of iteration is expressed by a simple variant of the tuple iterator: performing a computation C repeatedly N times is equivalent to performing C once for each one of the integers in the range: 1,2,3.. up to N. This range of values is expressed in SETL by means of the expression:

```
[1..N]
```

and thus the repeated computation of C is expressed as follows:

```
(FOR i IN [1..N] )
    C;
END;
```

The construct [1..N] looks like a tuple former, and indeed in contexts where a tuple is permissible, it is a valid tuple expression, as we saw in our discussion of tuple formers (2.7). But in an iteration this construct designates the range of values taken on by the loop variable in the course of the iteration. Note that an iteration variable appears here, just as it did in set and tuple iterations. This variable takes on the values specified by the range construct, in the order indicated, that is to say from 1 up to N in steps of 1.

Because of the importance of numeric iterators in programming, SETL provides a still more general form to describe them. We now proceed to explain this more general numerical iterator form.

3.3.4.2 The general form of the numerical iterator.

Any numerical iterator defines the sequence of integer values to be taken on by the iteration variable of a loop. The simple iterator form given above specifies the beginning (or lower bound) of the iteration to be performed as

1, and the end (or upper bound) as N. The step between successive values of the sequence iterated over is 1. In a more general numerical iterator, these three quantities: lower bound, upper bound and step, can be specified by means of expressions. To do so use the following construct:

[first, second .. last]

where first, second and last are integer-valued expressions. For example

```
[1,3..9]        specifies the sequence 1,3,5,7,9
[2,5..17]       specifies the sequence 2,5,8,11,14,17
```

As these examples indicate, the sequence iterated over is calculated as follows:

a) The lower bound is the first expression in the iterator.
b) The step between successive elements is the difference between the second expression and the first. If the second expression is missing, then, as in the examples of Section 3.3.4 then the step is understood to be 1.
c) Successive elements of the sequence are produced by repeatedly adding the value of the step, until we reach the value of the last expression.

This description immediately raises three questions:

i) What happens if the step is negative ?
ii) What happens is the upper bound is not in the generated sequence ?
iii) What happens if the step is zero ?

The answer to i) is what you would intuitively expect, namely: if the step is negative, then the elements of the sequence are produced in decreasing order. In that case, the third expression must be smaller that the first. For example, the iterator:

```
[10,8..0]      specifies the sequence 10,8,6,4,2,0
```

because the step is 8 - 10 = -2.

This form of the iterator is often used when the elements of a tuple must be processed in reverse order. For example, suppose that the elements of tuple T are numbers sorted in increasing order, and we want to list them in decreasing order, starting from the largest. The following loop will accomplish this:

```
(for i in [#T, #T-1..1])
    print(T(i));
end;
```

In this example, the first element of the sequence is given by the expression #T; the first value of the iteration variable -i- is therefore the index of the last element of T. The next value is #T-1, from which we conclude that the step for this sequence is -1. The last value iterated over is 1.

Next consider the second question raised above, namely: what if the final value appearing in the construct [first,second..last] is not in the generated sequence ? For example, what is the sequence generated by the following iterators:

[1,3..10]

and

[15,10..1]

The answer to this question is determined by the following rule: a sequence iterated over is generated by successive additions of the step to the first element. If the sequence is increasing (i.e. if the step is positive) we generate all numbers in the sequence which are smaller than or equal to the last element. If the sequence is decreasing, we generate all the numbers that are larger than or equal to the last element. Thus, for example,

[1,3..10]	specifies the sequence 1,3,5,7,9
[15,10..1]	specifies the sequence 15,10,5

What about [1,3..1]? According to the rule just stated, we start with 1. The step is 2. The next value in the sequence would be 3, but that is already greater than the stated upper bound of 1. Thus this iterator generates a singleton sequence, whose only element is 1. This leaves one final question: what is the meaning of the iterator if the step of the sequence is zero ? In that case, the convention used by SETL is that the iteration is empty, i,e. iterates over no values at all. A loop whose iterator has a step of zero is simply not executed. The following are examples of empty loops:

```
(FOR I IN [1,1..1000])
   PRINT('This message will never be seen');
END;

(FOR x IN {})
   PRINT('Nor will this one, because {} has no elements');
END;

(FOR i IN [])
   PRINT('Need we say more ?');
END;
```

The previous rule also answers another lingering question: what is the value of the loop variable on exit from a numerical loop ? We saw that in the case of set and tuple iterators, the loop variable became undefined on exit from the loop. In the case of numeric iterators, the value of the loop variable on exit is the first value in the sequence:

first, first + step, first + 2*step..

which lies outside of the specified range. If the step is positive, this means the first value of that sequence which is larger than the stated bound; if the step is negative, it is the first value which is smaller than the bound.

The preceding remarks apply (as you may have gessed) to the set- and tuple former constructs which build sequences of integers. In fact the iterators that we have been describing function in the same way in both contexts: when controlling loops with statement blocks, and when they control the construction of a composite object. This symmetry should be even clearer from the following code fragments, both of which build a tuple T by means of a numerical iterator:

```
(1)     T := [first, second..last];

(2)     T := [];

        (FOR x in  [first,second..last]) T with:= x;  END;
```

The action of the numerical iterator ' FOR i in [exp1, exp2..exp3]' can also be defined by the following 'low-level' code which uses labels and GOTO statements. (The intent of the GOTO statement, which is described fully in Sec.3.4, is to indicate the next statement in the program that should be executed following the execution of the GOTO itself).

```
            start := exp1;
            step  := exp2 - exp1;
            bound := exp3;

            IF step = 0 then GOTO quit_loop; END IF;
            i  := start;

    test_loop :
            IF (step > 0 and i > bound) OR
               (step < 0 and i < bound) THEN
                    GOTO quit_loop;
            END IF;
            .....           $ Body of loop.
            i +:= step;
            GOTO test_loop;

    quit_loop :
            .....           $ Statements following the loop.
```

3.3.5 Additional loop control statements: CONTINUE and QUIT.

The CONTINUE and QUIT statements increase the syntactive flexibility of SETL's loop constructs. Their syntax is simply:

CONTINUE *optional loop tokens*;

QUIT *optional loop tokens*;

In both cases, the *optional loop tokens* define the loop to which the intended action (continue the iteration, or quit altogether) refers. The actions caused by CONTINUE and QUIT are as follows.

a) When a CONTINUE statement is executed in the body of a loop, execution of the rest of the body is skipped, and the iteration proceeds to the next value of the loop variable. Thus, the loop:

```
(FOR x IN S | C(x))
    some statements...
END;
```

can be expressed as follows:

```
(FOR x IN S)
    IF C(x) THEN
        some statements..
    ELSE CONTINUE;
    END IF;
END;
```

b) Execution of a QUIT statement terminates the execution of a loop, and causes execution to continue from the first statement following the end of that loop. For example, consider the following fragment:

```
sum := 0;
(FOR x IN [1..100])
    sum := sum + x;
    IF sum > 10 THEN QUIT; END;
END;
print(sum);
```

This code fragment adds the integers in the range 1..100 until the sum is greater than 10. After 5 iterations through the loop, sum is 1+2+3+4+5 = 15, and at that point the QUIT statement is executed. The value printed is 15, and the 95 iterations that remain are simply not executed.

The CONTINUE statement might be typically used in a search loop, when an object x satisfying a property C(x) is to be found in some data aggregate S, and then processed in some way. When so used, the body of the loop is code that tests each element of S for the property C. It may be the case that we can determine that a given element y of S does not have the property C, even before completing the execution of the loop body. In that case, the CONTINUE statement allows us to avoid processing it, and proceed to the next element of S. We will see an example of such use below.

Like the CONTINUE statement, QUIT also appears typically in search loops. However, whereas CONTINUE usually bypasses unsuccessful cases, QUIT is used to signal that there is no need to continue with the iteration, either because the search has been successful, or because it has become clear that the search will in fact be unsuccessful even if the remaining elements are examined. In what follows we will see examples of both uses of QUIT.

c) When they are written without additional tokens, the CONTINUE and QUIT statements always refer to the innermost loop within which they appear. They also have an extended form, for example

(4) CONTINUE FOR x IN S;

which can be used to indicate which of the several nested loops within which a CONTINUE (or QUIT) statement like (4) appears, is to be CONTINUED. In this example, the loop meant is the innermost loop whose iterator starts with the tokens:

(FOR x IN S...

The same applies to sequences of tokens following a QUIT statement.

To illustrate the use of these statements let us return to the problem of producing a table of prime numbers. This time, we will write our program as a series of loops. Moreover, we will start with a simple solution to the problem, and improve this initial solution in order to develop more and more efficient versions of it. Our initial solution simply restates the definition of prime number: it is a number that has no factors except 1 and itself. In order to determine whether N is prime, we divide N by all numbers smaller than itself. If any of these divisions turns out to have no remainder, N is not prime, and we do not need to continue examining other divisors. If no division is exact, N is prime. Our first version reads as follows:

```
PROGRAM primes1;

N := 1000;                          $ The desired range.
primes := [];                       $ Sequence to be constructed.

(FOR num IN [2..N])                 $ Examine all numbers in the range

    (FOR factor IN [2..num-1])  $ Range of its possible divisors
        if num MOD factor = 0 then
            $ num has an exact divisor. Skip it.
            CONTINUE FOR num;
        end if;
    END FOR;

    $ If we reach this point, num is a prime.
    primes with:= num;

END FOR;
print('Primes in the range 1 to ',N, ': ');
print(primes);

end PROGRAM;
```

This simple program involves many redundant calculations, which we will proceed to discover and remove.

First, note that an even number (with the exception of 2) cannot be a prime number. There is therefore no need to iterate over all numbers in the range [2..N]. It is sufficient to consider only the odd numbers in that range. By the same token, these numbers can only have odd divisors. The outer loop should therefore have the range:

```
(FOR num in [3,5..N])
```

and the inner one

```
(FOR factor in [3,5..num-1])
```

This modification of the initial program makes it four times faster (only half as many operations are performed during each of the two nested levels of iteration.)

Next, note that to determine whether -num- is prime, we do not need to examine all its possible divisors: it is sufficient to examine its possible prime divisors, i.e. all prime numbers smaller than it. If we modify the inner iterator accordingly, we obtain the following program:

```
PROGRAM primes2;

N := 1000;                          $ The range.
primes := [2];                      $ The first prime.

(FOR num IN [3,5..N])
    (FOR factor IN primes)

        IF num MOD factor = 0 then
            CONTINUE FOR num;
        END IF;
    END FOR;
    primes with:= num;

END FOR;

print('primes in the range 1 to ', N, ': ');
print(primes);

END;
```

Out next improvement generalizes the observation that allowed us to eliminate all even numbers from consideration: whenever we find a new prime P, we can calculate all the multiples of P in the range 1..N and mark them 'not primes' so that we do not have to examine them for primality later on. The easiest way of acomplishing this is to keep a set of candidate numbers, from which we remove the multiples of each prime we find. This leads us to an improved program which reads as follows:

```
PROGRAM primes3;

N := 1000;
primes := [2];
candidates := {3,5..N}; $ At first, all odd numbers.

(FOR num in [3,5..N] | num IN candidates)
    (FOR factor IN primes)
        IF num MOD factor = 0 then
            CONTINUE FOR num;
        END IF;
```

```
        END FOR;

        primes with:= num;
        $ Now delete all multiples of num from the set of candidates
        (FOR multiple in [num, 2*num..n])
            candidates less:= multiple;
        END FOR;

    END FOR num;
    print('Primes in the range 1 to ', N, ': ');
    print(primes);

    END PROGRAM;
```

This suggests yet another substantial improvement to our program. We notice that whenever we examine -num- for primality, we will have already deleted from (-candidates-) all multiples of prime numbers smaller then -num-. Therefore, -num- is not a multiple of any of them, and it definitely IS the next prime. In other words, whenever we reach a number in the range 1..N which is still in the set of candidates, we know that that number is definitely prime, and the loop to find a factor for it is unnecessary. Our program now reduces to the following procedure known as the Sieve of Erastosthenes:

```
    PROGRAM primes4;

    N := 1000;
    primes := [2];
    candidates := {3,5..N}; $ At first, all odd numbers.

    (FOR num in [3,5..N] | num IN candidates)

        primes with:= num;
        $ Now delete all multiples of num from the set of candidates

        (FOR multiple in [num, 2*num..n])
            candidates less:= multiple;
        END FOR;

    END FOR num;

    print('Primes in the range 1 to ', N, ': ');
    print(primes);

    END PROGRAM;
```

Several small additional improvement to prime4 can still be made. Let us mention the following simple one: the set -candidates- may become empty before the outer iteration is completed, in which case all subsequent evaluations of the predicate: -num IN candidates- will fail. We can bypass these final useless iterations by adding the following statement immediately after the loop that eliminates multiples of the latest prime found:

```
IF candidates = {} then QUIT; END IF;
```

When a loop is exited by means of a QUIT statement, rather than after completion of its iteration, then the loop variable retains the value it had just before execution of the QUIT statement. This makes it possible to tell outside of the loop what was the last value of the domain of iteration that was examined. For example, in order to tell whether our last modification to primes4 was particularly useful, we could add the following statement on exit from the outer loop:

```
print('Last number examined: ', num);
```

In this case it turns out that 997 is a prime, so that testing to determine whether (candidates = {}) saves us only one check in the iterator.

3.3.6 Map iterators.

We have emphasized repeatedly that maps are sets. Hence to iterate over all the elements p of a map f we can simply write

```
(FOR p IN f) ...
```

In this iteration, the bound variable p is assigned successive elements of f, that is to say ordered pairs. If within the body of such a loop we wanted to refer to successive elements in the domain of f, we could 'unpack' p by writing:

```
(FOR p IN f)

    x := p(1);          $ x is in the domain of f
    y := p(2);          $ y is the corresponding point
                        $ in the range.
....
```

This same unpacking effect could also be obtained by placing a tuple assignment of the form:

```
[x,y] := p;
```

(See Sec.2.8) at the start of the body of the iteration or by changing the iteration header itself to read

```
(FOR [x,y] IN f) ...
```

Because of the importance of this type of iteration a still more elegant, map-like alternative notation is provided for it, namely

(5) `(FOR y = f(x))`

This form of iterator is called a map iterator. Note that both the variables x and y are bound by this iterator: x receives successive values taken from the domain of f, while simultaneously y is set to the corresponding range value f(x).

For example, suppose that f is the following map:

```
{ ['New York', 'Albany'],     ['California', 'Sacramento'],
  ['Massachusetts', 'Boston'], ['Illinois', 'Springfield'],
  ['North Dakota', 'Fargo'],   ['Idaho', 'Boise'] }
```

and that mid_west is the set:

```
{'Kansas', 'llinois', 'South Dakota', 'North Dakota', 'Michigan',
      'Iowa', Nebraska'}
```

then the following loop:

```
(FOR capital = f(state) | state NOTIN mid_west)
    print('the capital of ', state, ' is ', 'capital');
 END FOR;
```

will have the output:

```
The capital of New York is Albany
The capital of California is Sacramento
The capital of Idaho is Boise
The capital of Massachussetts is Boston
```

The syntax appearing in (5) can also be used for tuple iterators. If T is a tuple, then the iterator

```
(FOR comp = T(i))
```

assigns the integer values 1,2..#T to -i-, and simultaneously assigns the values of the corresponding components of T to -comp-. The advantage of this form over the simple tuple iterator is that it makes the index of each component available at the same time as the component. (The use of a syntax like that of map iterators for tuple iterators once again underlines the logical similarity between tuples and maps: tuples are very similar to maps whose domain is a set of integers.)

The iterator (5) can only be used for single valued maps, and the system will generate a run-time error if we attempt to use it on a multivalued map. To iterate over a multivalued map, the following form is provided:

```
(FOR s = f{x})                    (6)
```

Like (5), this construct, sometimes called a multivalued map iterator, controls both the values of x and s. The variable x receives successive values from the domain of f, and s becomes the corresponding image set of x, that is to say f{x}. For example, let f be the map

```
{[i,j] : i in [1..4], j in [1..4] | i > j}
```

Then the iteration

```
(FOR s = f{x} | ODD #s)
print(s, 'is the image of ', x);
```

```
END;
```

will produce the following output:

```
{1,2,3} is the image of 4
{1} is the image of 2
```

3.3.7 Compound iterators.

A compound iterator is a useful shorthand notation to describe nested iteration loops. For example, the code fragment:

```
(FOR x IN S1)
    (FOR y IN S2)
        .......
    END;
END;
```

can be written as follows:

```
(FOR x IN S1, y IN S2)
    .......
END;
```

Any number of nested loops can be combined in this fashion. A single END statement closes all of them. The iterators in a compound iterator are understood to be nested from left to right. The rightmost iterator in the compound is the innermost; its loop variable changes most rapidly.

All iterator forms can appear in a compound iterator: set and tuple iterators, numeric iterators, map iterators. For example:

```
(FOR x IN S, y IN [1..x-1], z = f(t)) ....
```

Compound iterators can also have a 'such that' clause. Such a clause is understood to apply to the innermost iterator in the compound, that is to say, this clause is evaluated for every assignment to the innermost loop variable.

CONTINUE and QUIT statements appearing within a compound iterator apply uniformly to the outermost iterator therein: there is no way to continue or quit any of the inner members of the compound. (If it is necessary to do so, the iterators should be written in the usual nested form).

3.3.8 The general loop construct.

Each of the iterators discussed so far generate a sequence of values: the successive elements of a set, the components of a tuple, the characters of a string. We have seen how iteration loops are described by means of such iterators: the body of a loop is executed once for each value that appears in the generated sequence. Different kinds of loop constructs called WHILE and UNTIL loops, are used to describe computations that repeat until a desired state of affairs is reached, rather than according to some preset sequence of values. For example, we may want to process input data which is to be read from a file, but we may not know how many items are actually

present on the file. In this case, we need to express the following intent: "Process the input as long as there is data to process. " Numerical analysis furnishes a second example. Many numerical problems have the following general flavor: find a sequence of better and better approximations to a desired value (for example, to the root of an equation) and stop when the answer is 'close enough'. (Close enough usually means that rather than looking for an exact answer, we are satisfied with an answer which differs from the exact one by a very small number, say 1E-7.) In these cases, we generally cannot state in advance how many times the loop body may have to be repeated. For use in these situations, SETL makes a very general loop construct available. The simplest form of this general construct is the 'indefinite loop', whose syntax is as follows:

```
LOOP DO
   block
END;                                $ Or END LOOP;
```

As with the simpler iterator forms, the keywords LOOP and DO can be represented by parentheses. Thus, the indefinite loop can also be written:

```
() block END;
```

An indefinite loop specifies that the loop body should be repeated 'forever'. This is clearly an overstatement: the computation will have to finish somehow! In fact, an indefinite loop can be terminated either by using a QUIT statement, by a CONTINUE statement which refers to an enclosing loop, or by means of a GOTO to a label which is outside the loop body.

The indefinite loop is not used very often, because ordinarily the condition under which it will terminate execution can more clearly be expressed by means of extremely useful loop forms, namely the WHILE and UNTIL LOOPs. Let us now examine these.

3.3.8.1 The WHILE loop.

A WHILE loop is written as follows:

```
LOOP WHILE condition DO
    block
END;
```

or equivalently

```
(WHILE condition)
    block
END;
```

Execution of such a loop proceeds as follows:

The condition is evaluated. If its value is TRUE, then the loop body is executed. After each execution of the body, the condition is evaluated anew, and as long as it yields TRUE, the body continues to be executed again. As soon as the condition becomes FALSE, looping ends, and execution proceeds with the first statement that follows the loop.

If the first evaluation of the condition yields FALSE, then the loop body is not executed at all. If follows that a WHILE loop can be executed zero or more times.

Let us look at some examples. As we have already seen, the processing of a stream of data received from input is a typical case: suppose that we want to read a list of names and print those that start with 'A'. We do not know the number of items in the data stream, and it may even be that there are none. Fortunately, the SETL system uses a very simple convention to indicate that data has been exhausted. When we attempt to read data from a file, but have reached the end of the file, the READ statement yields OM. Thus, the following simple code fragment can be used to handle a stream of input data and stop when the end of the data has been reached:

```
read(name);                          $ Get first name from input file.
count := 0;
(WHILE name /= OM)                   $ As long as we read something

    IF name(1) = 'A' THEN
        print(name);
        count +:= 1;
    END IF;

    read(name);          $ Acquire next data item from input.

END WHILE;

print(count, ' names starting with A were found');
```

Note that in this code we execute one READ statement before the loop, t 'prime' the loop, so to speak. Doing this ensures that -name- receives a value before the first evaluation of the WHILE condition. If the input file was not empty, then -name- is not OM, and the body of the loop is executed. If the input file was empty, then -name- is OM, and the loop is bypassed altogether. At the end of each execution of the loop body, we perform another READ operation. As long as something is read, the loop will be executed again. As soon as the strem of input data is exhausted, the READ statement will yield OM, the WHILE condition evaluate to FALSE, and execution of the WHILE loop will terminate. Program execution will then proceed to the statement following the loop, which in the case above is the one that prints the little statistical report on the data.

Our next, more complex example is motivated by the following practical problem. Suppose that the catalog of a school specifies a set of prerequisites for each course that is offered. That is to say, for each course C, it specifies a set of courses which the student must have taken before being allowed to take C. Needless to say, the prerequisites of C often have further prerequisites of their own, and we will sometimes want to know the full set of courses that have to be taken before C is tackled. These include the prerequisites of C, the prerequisites of those prerequisites, and so on. Let us assume that the map -prerequisites- contains the standard information that appears in the school catalog, that is to say the list of immediate prerequisites of each course C. Then the desired set can be obtained as follows:

```
P := prerequisites(C);          $ get the 'immediate' pre-
                                 $ requesite for the course C

all_P := P;                     $ initialize the set we aim
                                 $ to build

(WHILE P /= {})                  $ as long as there is some pre-
                                 $ requisite that has not been
                                 $ processed.

   course FROM P;                $ take one of them.

    all_P WITH:= course;         $ Add to full set of pre-
                                 $ requisites.

    P +:= prerequisites(course); $ add all the prerequisites
                                 $ of P to the set

    END WHILE;
print('Before taking ', C,' the following must have been taken');
print(all_P);
```

This example deserves careful study, because it embodies a very common type of program schema, sometimes called the use of a 'workpile'. The set P originally consists of the immediate prerequisites of C. Each of these is placed in all_P, which is to be built up to the full set of prerequisites we are gathering, and each of their prerequisites in turn must be placed in all_P, and also into the set P, to see whether further prerequisites are implied by them. The process terminates when we reach courses that have no prerequisites at all (there must be some of those!). The 'workpile' set P shrinks with each execution of the FROM statement, but can increase again with the addition of the prerequisites of the course we have just extracted from P. 'Workpile' algorithms of this kind typically involve WHILE loops.

3.3.8.2 The UNTIL loop.

The syntax of the UNTIL loop is similar to that of the WHILE loop. We write:

```
LOOP UNTIL condition DO
   block
END;
```

or equivalently

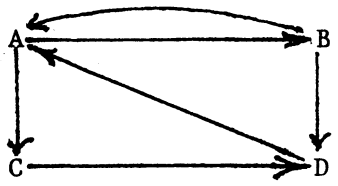
```
(UNTIL condition)
    block
END;
```

An UNTIL loop is executed as follows:

The body of an UNTIL loop is always executed at least once. After it is executed the loop condition is evaluated. If it yields TRUE, then execution proceeds to the first statement following the loop. If it yields FALSE, the body of the loop is executed again. We can therefore say that the test of a

WHILE loop is performed at the beginning of the loop body, while the test of an UNTIL loop is performed at the end of the loop body. Note also that the body of an UNTIL loop is always executed one or more times, in contrast to a WHILE loop, which may not be executed at all.

As an example, let us consider the problem of finding the smallest number of steps that can take us from one point in a graph to another. In order to tackle this problem we must say a word about graphs, and about the ways in which they can be described in SETL. A graph consists of a set of vertices, and a set of edges which connect the vertices. Edges of a graph can be represented in SETL by ordered pairs, whose first component is the starting vertex for an edge, and whose second component is the arriving vertex for that edge. For example, the simple graph:



is described by the following set of pairs (i.e. edges):

{[A,B], [B,A], [A,C], [C,D], [B,D], [D,A]}

Since in SETL a set of pairs is at the same time a map, we can also regard this representation as a 'successor map' (also called an 'adjacency list') whose domain is the set of vertices of the graph. Then, for each vertex V, the value of the mapping succesor{V} is the set of vertices that are reachable from V by means of some edge that starts at V. For example, in the graph above, successor{B} is the set {A,D}, because of the existence of edges from B to A and D.

Using this bit of notation, our problem can be stated as follows: given a graph G, described by means of its set of edges, and given two vertices s(ource) and t(arget) find the length of the shortest path between s and t, i.e. the smallest number of edges that must be traversed in order to go from s to t. If we do not know a priori what path to take, we may have to explore a substantial number of paths starting from s, until we find one that reaches t. A possible way of organizing this exploration is to find all the vertices that can be reached from s in one step, two steps, etc., until we reach t. Our problem will therefore be solved by the following:

```
seen := {s};                    $ The set of vertices already
                                $ reached.
length := 0;                    $ The length of the path so far.
```

```
(UNTIL t IN seen)
    $ Add to -seen- all the vertices that can be reached by
    $ following one more edge from vertices already reached.

    (FOR v IN seen)
        seen +:= successor{v};
    END FOR;

    length +:= 1;

END UNTIL;

print('There is a path of length ', length, ' from ', s,
                          ' to ', t);
```

Various shortcomings of this code are easily noted: for example, what if our graph is such that there is no path from s to t ? As written, our algorithm will iterate indefinitely, and the condition -t IN seen- will never be met. We will endlessly retrace the edges that lead out of the vertices already reached. In order to prevent this behavior, we can modify our algorithm, so that at each step -seen- contains only those vertices that have not been reached on previous steps. This can be achieved as follows:

```
seen := {s};                        $ The set of vertices reached
                                    $ at each step.
reached := {s};           $ The set of all vertices
                                    $ reached so far.

(UNTIL t IN seen OR seen = {})

   $ We collect the new vertices reachable from the latest set,
   $ which were not reached previously.

   seen :=+/{successor{v}: v IN seen} - reached;

   reached +:= seen;

   length +:= 1;

END UNTIL;

IF seen = {} THEN
    print(t, ' is not reacheable from ', s);
ELSE
    print('There is a path of length ', length, ' from ', s,
                          ' to ', t);
END IF;
```

See section 4.3.1 for a further example continuing this theme.

3.3.8.3 The general LOOP construct.

The indefinite loop, the WHILE, and the UNTIL loops are all simple cases of a more general SETL loop construct, whose impressive full syntax is as

follows:

```
LOOP
    INIT blocki          $ Loop initialization statements.
    DOING blockd                $ Step statements at startof loop.
    WHILE testw                 $ Termination test at start of loop.
    STEP  blocks                $ Step statements at end of loop.
    UNTIL blocku                $ Termination test at end of loop.
    TERM  blockt                $ Loop termination statements.
DO
    blockb                      $ Body of loop
END;                            $ Or END LOOP tokens;
```

iteration

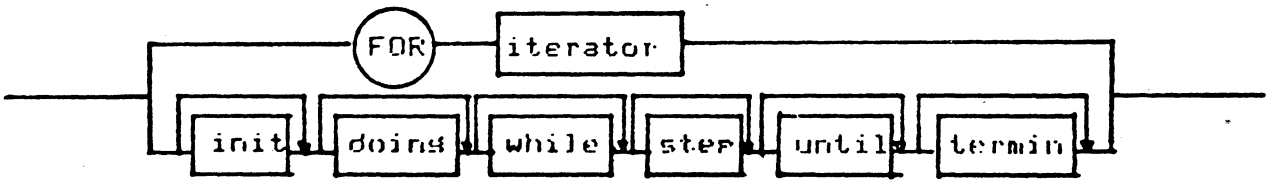


Fig 3.6: General iteration syntax diagram

You will notice that WHILE and UNTIL clauses are both included in this very general loop form. Its full structure is complex. However, as you may have surmised from the previous sections, every single clause in this loop construct is optional ! If we leave all of them out, we obtain the indefinite loop: LOOP DO ... The WHILE and UNTIL loops are obtained by keeping only one of the loop clauses. To explain the full construct, we must now describe the purpose of the remaining four clauses in it: INIT, DOING, STEP, and TERM.

3.3.8.4 The DOING and STEP clauses.

The reader will have noticed that the body of WHILE and UNTIL loops always includes at least one statement that affects the value of the boolean condition that controls the execution of the loop. In example 1, above, this was the statement: read(name); which can set name to OM and terminate loop execution. In example 2, it is the statement

```
P +:= prerequisites(course);
```

that affects the boolean condition controlling the loop. The readability of a loop is often improved by indicating 'housekeeping' actions directly in the loop header, close to the condition that governs loop execution. This can be done using the DOING and STEP clauses of the loop construct.

a) If a DOING clause appears in a loop construct, then the block of statements labelled by the keyword DOING is executed each time through the loop, before the loop body is executed, and also before the WHILE condition (if present) is evaluated.

b) If a STEP clause appears in a loop construct, then the block of statements labelled by the keyword STEP is executed each time through the loop, immediately after the loop body has been executed, and before the UNTIL condition (if present) is evaluated. For example, using the DOING clause, our first example can be rewritten as follows:

```
LOOP WHILE name /= OM STEP read(name) DO
....
END LOOP;
```

Similarly, example 4 can be rewritten as follows:

```
seen := reached := {s};

LOOP STEP reached +:= seen;
          length +:= 1;
          UNTIL t in seen DO
...
END LOOP;
```

All the the numeric iterators which we examined in Section.3.3.5 can be described using WHILE and UNTIL statements with STEP clauses. For example, the following loops are all identical in their effect:

```
(FOR i in [1..100]) .. END;                                    (1)

i := 1;
(WHILE i <= 100 STEP i +:=1;) .. END WHILE;                    (2)

i := 1;
LOOP UNTIL i = 100 STEP i +:= 1; DO .. END;                    (3)

i := 0;
(WHILE i <= 99 DOING i +:= 1;)                                 (4)
 ...
END WHILE;
```

Choosing between these constructs is a matter of style. If the iterator is numeric, and the associated actions are arithmetic increments, then (1), which is simplest, is to be prefered. The reader will find it instructive to transcribe the various forms of the numeric iterators into loop constructs that use WHILE, UNTIL, STEP and DOING clauses. (See exercises XXX-YYY).

3.3.8.5 The INIT and TERM clauses.

Th INIT and TERM clauses of the loop construct allow us to specify initialization actions and termination actions to be performed upon entry and exit from the loop.

a) If the INIT clause is present, then the block of statements labelled by the INIT is executed once before any execution of the loop body, and before evaluation of the WHILE clause (if present).

b) IF the TERM clause is present, Then the block of statements labelled by the TERM keyword is executed once on exit from the block, after evaluation of the UNTIL clause (if present).

To summarize, the precise effect of the complete loop construct:

```
LOOP INIT  blocki
     DOING blockd
     WHILE testw
     STEP  blocks
     UNTIL testu
     TERM  blockt
DO
    block
END;
```

can be described by the following equivalent sequence of statements:

```
          blocki            $ The INIT block.
start:
          blockd            $ The DOING block.
          if NOT testw then       $ The WHILE condition.
              GOTO term;
          end if;
          block             $ The actual body of the loop.
step:     blocks            $ The STEP block.
          if testu THEN     $ The UNTIL condition.
              GOTO term;
          end if;
          GOTO start;             $ To continue looping.
term:
          blockt            $ The TERM block.
```

The labels appearing in this code segment also allow us to give a simple definition of CONTINUE and QUIT statements in a loop construct. The statement

```
CONTINUE;
```

is equivalent to the statement

```
GOTO step;
```

where -step- is the label that precedes any code taken from the STEP clause. The statement:

```
QUIT;
```

is equivalent to

```
GOTO term;
```

where term is the label that precedes the code associated with the TERM

clause.

3.4 The GOTO and STOP statements.

In the example given just above, and also in the previous chapters, we have several times made use of the notion of a GOTO and the concept of a label.

It is time to describe this very basic statement carefully. A GOTO statement changes the flow of program execution in the most direct fashion. When we execute the statement:

```
GOTO there;
```

then execution of the program passes immediately to the statement marked by the label -there-. (A SETL label is simply an identifier followed by a colon.) Any executable statement can be labeled, and any number of labels can appear before a statement.

The GOTO statement has come to be regarded as a dangerous construct, whose use should be avoided, and some programming languages exclude this statement altogether. While avoiding this puritanical approach, we stress that the GOTO statement is only rarely useful, and that one should strive to describe control flow using the safer constructs described so far: conditionals, case statements and loops, but not GOTO's.

Reservations concerning unrestricted use of the GOTO rest on sound and pragmatic grounds. Programs that depend heavily upon the use of GOTO's are hard to read and to understand, difficult to modify, error-prone and thus dangerous. Heavy use of labels and GOTO's obscures the logical structure o a program. In particular, when backward jumps appear in the middle of a large program, their intent is obscure, and the purpose of the code is therefore harder to comprehend.

There are however a few cases in which the GOTO statement is useful. The most common of those cases has to do with abrupt exits from a sequence of related code fragments. If these all test for some common kind of error, it may be appropriate to place a label past the end of all these fragments, and to GOTO this label if an error is detected. This is most commonly in this guise that the GOTO will be seen in this book. Note that exit from loops is clearly described by QUIT and CONTINUE statements, which should always be preffered to GOTOs and labels.

SETL imposes certain restrictions on the position of labels and of GOTOs that refer to them. These restrictions as the following:

a) A GOTO lying outside of a loop construct cannot refer to labels that appear within the body of the loop.

b) A GOTO can only refer to a label that appears within the same procedure or main program as the GOTO. (See Chapter IV for a discussion of 'procedures' and the notion of a 'main program').

c) Label names are local to the procedure in which they appear. (See

Sec.4.2 for information on 'scoping rules').

The STOP statement

The STOP statement is simply used to terminate execution when for some reason your program has decided that it cannot go on. This statement can be used either in your 'main' program or in any 'subroutine' or 'function' (see Chapter 4.) A typical example of its use might be

```
IF x > 0 AND x*x < 0 THEN
   print('*** SITUATION ALL FOULED UP. PROGRAM CANNOT CONTINUE ***');
   STOP;
ELSE
   .......   $ do whatever needs doing
```

Of course, your program will always stop by itself when it has executed the last statement of your (main, see Chapter 4) program. So no STOP statement is needed there (even though it does no harm to put one in.)

3.5 Programming example: an interpreter for a simple language.

One of the most typical uses of the CASE statement is programming an 'interpreter'. An interpreter is simply a program that executes sequences of commands written in some formalized language. An interpreter works by reading one command at a time, executing it, and then reading the next command, etc. Interpreters serve as an obvious means of creating special-purpose languages, and we will say more about this at the end of this section; but first we will present an example of an interpreter. This will make use of most of the control structures that we have examined so far in this chapter.

We will write an interpreter for the so-called 'Turtle language' used in a popular system for grade-school computer education. The Turtle language consists of a series of commands that control the motion of a 'Turtle' on a screen or on a sheet of paper. The motions of the turtle generate a picture, and the purpose of the interpreter is to read a series of commands in Turtle language and construct the corresponding picture. The position of the Turtle at any given time is described by its coordinates, and its direction of motion. The turtle can be commanded to move forward a certain number of steps, turn left or right, and put its pen down (to draw) or up (to move without drawing a line.) The full list of commands and their syntax is the following:

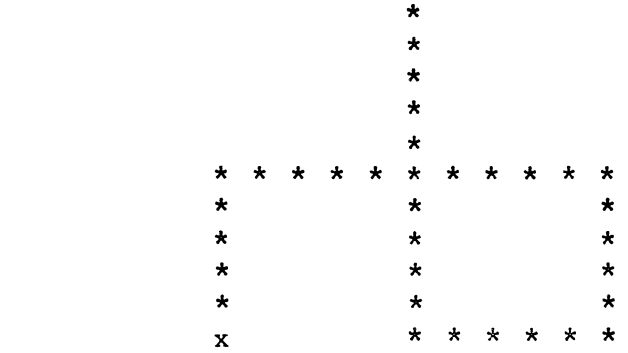
```
FORWARD N              -- Move forward N steps.
RIGHT                  -- Turn right from current direction of motion
LEFT                   -- Turn left from current direction of motion.
PEN_UP                 -- Move without leaving a trace.
PEN_DOWN               -- Draw every motion.
DRAW                   -- Display picture of motions so far.
END                    -- Terminate picture, draw it and stop.
```

For example, the sequence of commands:

```
PEN_DOWN
FORWARD 5
```

```
RIGHT
FORWARD 10
RIGHT
FORWARD 5
RIGHT
FORWARD 5
RIGHT
FORWARD 10
DRAW
```

generates the following picture:



Construction of a Turtle language interpreter.

The preceding description of the meaning and purpose of each Turtle language command should make it clear that our interpreter will consist largely of a simple CASE statement, each of whose options correspond to one command in the Turtle language. That is to say, the basic structure of the interpreter will look as follows:

```
            CASE command of
    (RIGHT'):  .....
    ('LEFT'):   .....
    (FORWARD'): ....
  etc.
```

Of course, we have to fill the dotted sections with an exact description of the actions that represent the corresponding motion of the turtle. This requires that we decide on how to represent the picture being drawn, and also the position and direction of motion of the turtle at each step.

First let us examine the matter of picture representation. In order to keep our task simple, we asume that the track of the Turtle will be displayed by means of PRINT commands. Each PRINT statement generates one line of output, and it is reasonable to describe the picture as a sequence of lines. To make matters definite, we must choose the height and width of the picture: We let that be 50 by 50, so that it can fit easily on a simple page of printed output. This size will not change during execution of the program so we just initialize the picture to be an array consisting of 50 strings of length 50, consisting only of blanks:

```
picture := 50 * [ 50 * ' '];
```

Notice the double use of the replication operation '*': the expression 50*' ' yields a string of fifty blanks; The brackets around this expression give us a tuple whose only element is such a string; and the outer replication operation yields a tuple with 50 elements, each of which is a blank string.

Note that this is not the only possible way of representing the picture.(Try to think of some alternative representations). However we shall see that this choice simplifies the creation and display of the picture.

The position of the turtle at each step is defined by giving a line and a character position on the line. If we think of each line as drawn horizontally accross the picture, then the choice of [row, column] to designate the turtle position imposes itself. In our simple interpreter the turtle can move in one of four directions, which we can label 'NORTH', 'EAST', 'SOUTH' and 'WEST', with the usual (Northern Hemisphere) convention that north is up. We choose to start the turtle on its trek from the lower left-hand corner of the picture, facing north.

Next let us sketch the actions performed upon each Turtle command. The turning commands: RIGHT and LEFT, are the simplest: they only change the direction of motion of the turtle, not its position, and they do not add anything to the picture being drawn. We have chosen to implement those commands simply by looking up the direction that lies to the right or left of the present direction of motion. This lookup operation uses SETL maps.

The pen commands: PEN_UP and PEN_DOWN, affect neither the position nor the direction of motion of the turtle. We describe their effect using a boolean variable called -tracing-, which is interrogated whenever the turtle actually moves.

The only non-trivial command is -FORWARD N- where N is some positive integer. This command alters the position of the turtle, and produces a segment of the picture if the -tracing- indicator is TRUE. Clearly the action of FORWARD depends on the current direction of motion. If the turtle faces east, the motion will be to the right, along a line or row. The same is true if the turtle faces west. On the other hand, if the turtle faces north or south, then its motion is along a column, and its row position is altered. The forward statement is therefore best described by a CASE statement. Let -distance- designate the extent of the specified forward motion, and let new_row, new_col be the coordinates at which the turtle finds itself after the motion. Then the effect of FORWARD can be described as follows:

```
CASE direction of

    ('NORTH'): new_row := row - distance;
               new_col := column;

    ('EAST'):  new_col := column + distance;
               new_row := row;
                .....

etc.
```

Finally, how is the picture itself to be created ? We want to fill in the trajectory described by the turtle using some printable character, say the asterisk: '*'. After each FORWARD command, we want to place asterisks along the line from [row, column] to [new_row, new_column]. This is simple if the motion is horizontal, i.e. new_row = row, since in this case the line to be drawn is a part of the current row. If we recall that the picture is described by an array of horizontal lines or rows, then it is clear that the line on which the turtle is currently moving is given by: picture(row). The motion of the turtle fills a substring of this row, and in the case of Eastward motion this can be expressed as follows:

```
picture(row)(col..new_col) := distance * '*';
```

Westward motion is equally simple to describe. North-South motion is a trifle harder to handle. In such a motion, the turtle stays on the same column but crosses several rows. The line it traces has one character on each row traversed. We lay down the trace as follows:

```
(FOR i in [row..new_row])
    picture(i)(column) := '*';
end;
```

Finally, we want our interpreter to read any number of turtle commands, and we do not know a priori how many there will be. We therefore enclose our basic case statement in another loop, this one bracketed by the lines:

```
LOOP DO
```

and

```
END LOOP;
```

Finally, the statement:

```
STOP;
```

which our interpreter must associate with the END command, will terminate interpretation.

```
PROGRAM TURTLE;
 .s
right := {['NORTH', 'EAST'], ['EAST','SOUTH'], ['SOUTH', 'WEST'],
          ['WEST',  'NORTH'] };

$ The map giving the direction to the left of any direction is obviously
$ the inverse of the -right- map.

left := { [d1,d2] : [d2,d1] in right};
 .s
picture := 50 * [50 * ' '];
 .s
$ Initially the turtle is at the lower left-hand of the picture,
$ facing north.

direction := 'NORTH';
row := 50;
column := 1;
```

```
tracing := FALSE;

LOOP DO                          $ Main loop of the interpreter.

     read(command);

         CASE command of

     ('RIGHT'): direction := right(direction);

     ('LEFT'):  direction := left(direction);

     ('PEN_UP'): tracing := FALSE;

     ('PEN_DOWN'): tracing := TRUE;

     ('DRAW','END'): (FOR line IN picture)
                       print(line);
                     END;

                     picture := 50 * [50 * ' '];
                     IF command='end' THEN stop; END IF;

     ('FORWARD'): read(distance);

                  CASE direction of

                  ('NORTH'): new_row := (row - distance) MAX 1;
                             new_col := column;

                  ('EAST'):  new_col := (column + distance) MIN 50;
                             new_row := row;

                  ('WEST'):  new_col := (column - distance) MAX 1;
                             new_row := row;

                  ('SOUTH'): new_row := (row + distance) MIN 50;
                             new_col := column;
                  END CASE;

                  IF tracing THEN

                     IF new_row = row THEN

                        $ Find first and last column needed for tracing.
                        min_col := column MIN new_col;
                        max_col := column MAX new_col;

                        picture(row)(min_col..max_col) :=
                                        distance * '*';

                     ELSE

                        $ Find first and last  row.
                        min_row := row MIN new_row;
```

```
                        max_row := row MAX new_row;

                        (FOR r IN [min_row..max_row])
                           picture(r)(column) := '*';
                        END;

                    END IF;
                END IF;

                row := new_row;
                column := new_col;

    ELSE print('INVALID COMMAND: ',command);

    END CASE;

END LOOP;

END PROGRAM turtle;
```

Several additional details of this program deserve notice:

a) Two Turtle commands produce an actual drawing: the DRAW command, and the END command. It is natural to place both commands in the same CASE tag, and add an additional simple check, made after the picture has been produced, to determine whether the program should stop.

b) We all make mistakes, and the interpreter should be prepared to receive less-than perfect instructions. What should the interpreter do, for example, with the commands:

```
FORWARD 10
RIGHT
PEN_UP
```

and so on ? In this program we have chosen to notify the user that a command just read is not part of the known set of Turtle commands. This is the purpose of the ELSE clause of the CASE statement. A more ambitious program might try to recognize misspellings of the known commands, accept abbreviations for them, accept upper- and lower-case names for commands, and so on. Some of these extensions are pursued in the exercises below.

c) A different sort of error is exemplified by the command:

```
forward 200
```

Which attempts to move the turtle beyond the bounds of the picture. In the program above, we have made sure that the values of new_row and new_col are always in the range 1 to 50.

c) The printer is not the best device on which to display a picture. If you run the program as written, you will notice that the separation between successive lines is greater than that between successive characters on a line. As a result the picture looks cramped in the horizontal

direction. A more aesthetic result is obtained if we count each horizontal step as two characters, or always add a blank between horizontal characters. This modification is left to the reader.

3.5.1 Various elementary sorting techniques.

Sorting is the problem of taking a set or tuple of items which (like integers, real numbers, or strings) can be compared to one another, and putting them in order. Dozens of interesting ways of using a computer to sort are known, and a few of the more interesting high-efficiency sorting techniques will be presented in later chapters. In the present section, we present only some very simple sorting methods, which serve to illustrate various control structures discussed in this chapter. The first and simplest of these, the so-called bubble-sort method, sorts a tuple. It works simply by scanning the tuple for adjacent components which are out of order, and interchanging them if they are found. In this way, large items 'bubble up' to their proper position in the tuple. When no out-of-order pairs remain, the tuple is sorted.

In SETL this is simply

```
(WHILE EXISTS i IN [1..#t-1] | t(i) > t(i+1))
    [t(i),t(i+1)] := [t(i+1),t((i)]; $ interchange the items.
END WHILE;
```

The bubble-sort procedure has a number of interesting variants. In one of them, we simply sweep repeatedly through the tuple, interchanging all pairs of adjacent items which are out or order.

If we perform this sweeping operation at least as many times as the tuple has components, all items will be swept into their proper positions, since even if the smallest item originally came last it will have time to move down to the first position in the tuple.

We can express this 'sweeping' procedure as

```
(FOR number_of_times IN [1..#t])
    (FOR i IN [1..#t-1] | t(i) > t(i+1))
        [t(i),t(i+1)] := [t(i+1),t(i)]; $ interchange
    END FOR;
END FOR;
```

This can also be put more succinctly as

```
(FOR number_of_times IN [1..#t], i IN [1..#t-1])
    [t(i),t(i+1)] := [t(i+1),t(i)];
END FOR;
```

A very different sorting method is to search repeatedly for the minimum element of a tuple, put it at the end of a new tuple which is being built up, and delete it from the original tuple. This is called the selection sort method. It can be written as

```
newtup := [ ];    $ initialise tuple to be build up.
```

```
(FOR i IN [1..#t])

    min_till_now := t(1); min_place=1;
        $ save minimum element scanned, and its location

    (FOR j IN [2..#t] | t(j) > min_till_now)

        min_till_now := t(j); $ save value of newly found minimum
        min_place := j;  $ save position of new minimum

    END FOR;

    new_tup(#new_tup) := min_till_now;  $ put minimum element at
                                        $ end of new tuple

END FOR i;
```

Beyond the methods shown above, you will find that it is instructive to review all the ways you can think of to sort a deck of cards by hand, and to express these hand-sorting techniques in SETL.

3.6 Exercises

Ex. 1 A set of Markov productions is an ordered collection of rules of the form s1>s2, where s1 and s2 are both character strings, neither of which contains the character '>'. The string s1 is called the left side of the production s1>s2, and the string s2 is called its right side. To apply such a set of productions to a string s, one searches through s, looking for a substring which coincides with the left-hand side of some production; if any such production is found, this substring is replaced by the right-hand side of the production.

Write a 'Markov production interpreter' program which reads in a set of Markov productions and a string s, and then applies n successive productions to s, displaying the result every m steps.

Ex. 2 How would you express a FOR loop of the form

```
(FOR n IN [1..k]|C(n))..
```

in terms of a WHILE loop? What about FOR loops of the form

```
(FOR x IN t|C(x))...
```

and

```
(FOR x =t(i)|C(x))...
```

where t is a tuple?

Ex. 3 Write a program which will compare two poker hands (each consisting of five cards) and decide which of the two is the winning hand according to the values of Poker.

Ex. 4 Write a program which prints all the numbers from

2 to 100 together with their prime factorizations. The first three entries printed should be

```
2     [2]
3     [3]
4     [2,2]
etc.
```

3.7. Reading and Writing Data

We have been using the two basic Input-Ouput commands, READ and PRINT, which allow a SETL program to communicate with the rest of the world, informally till now. Now let us discuss them more systematically. (However, we pospone discussion of the more elaborate SETL input/output features, such as READA, PRINTA, GET, PUT,etc. to Section 8.1.)

To produce printed output (or, in the case of an interactive run from a terminal, to send output to the terminal), the PRINT statement is used. This has the form

PRINT(expl,exp2,...,expk),

where each of expl,...,expk is an expression. Any valid expression can appear in a PRINT statement, and any valid SETL value, including boolean values and atoms (see Section 5.3) can be printed. In particular, sets and tuples can be printed. Thus it is perfectly acceptable to write

```
PRINT (2+2, {1,2,3}, [{1},{{2}},[{3}]], 'HELLO THERE');
```

The output produced by this PRINT statement will look like

```
4   {3  1  2}    [{1} {{2}} [{3}]]     HELLO THERE
```

This example illustrates several details concerning the PRINT primitive:

(a) Expressions are evaluated before being printed.

(b) The elements of sets are grouped within set brackets, and tuple components are grouped within tuple brackets. For ease of reading, set elements and tuple components are separated by blanks rather than by commas (even though this can lead to ambiguities when structures containing strings are printed).

(c) Strings are printed without quotation marks, e.g. when we print the constant 'HELLO THERE' only the characters

HELLO THERE

appear in the output file.

(d) Since sets have no particular order when sets are printed, their elements can appear in any order.

(e) Integers and floating point numbers are printed in standard decimal formats. Their representations require a number of characters defined by their size and nature. Floating point numbers are always printed in exponential form with a fixed number of decimal places, e.g. 2.3 is printed as

2.300000E+00.

(f) Other kinds of SETL values will be represented by strings formed accordind to somewhat arbitrary rules. The undefined atom OM is printed as

*

The boolean values TRUE and FALSE are printed as #T and #F respectively. Atoms (see Section 5.3) are represented by strings of the form #nnn, where nnn denotes the integer 'serial number' of the atom. Note that these rules inevitably lead to a degree of ambiguity, e.g. the output produced by

```
print([OM, '*']);
print([TRUE,'#T',FALSE,'#F');
[*  *]
[#T  #T  #F  #F].
```

(g) Since sets are not printed in any particular sorted order, it can be hard to locate elements in the printed representation of sets, especially large sets.

(h) A single print statement (even a print statement with many arguments) will always try to put all the output which it produces on a single logical 'line' of output. If the value or values to be printed are too large and complex to fit on a single line, they will be printed on as many lines as necessary. When this happens, the points at which one physical line of print Ends and the next begins will fall haphazardly, and it can be something of a trial to read the resulting output.

(i) Each print statement starts a new logical line. A parameterless print statement can be used to generate a blank print line. Thus, whereas

```
print('AA','BB','CC');
```

will produce the output

```
AA  BB  CC,
```

the command

```
print('AA'); print('BB'); print; print('CC');
```

will produce the output

```
AA
BB

CC
```

(j) As illustrated by the preceding examples, successive output items produced by a single print statement are separated by a few blanks but do not start a new line.

The SETL print facility is quite easy to use, but does not produce output comparing in elegance with the formattted output generated by programs written in various other languages, especially languages such as PL/I or COBOL, which have something of a commercial orientation. To produce more elegant formulated output in SETL, it is necessary (albeit easy to make use of string primitives which the language provides (see Sections 2.3.3 and 5.2.) These allow one to build up output strings of arbitrary format and complexity. Note in particular that the STR operator produces the very same string representation of a value that the -print- command would print, but makes this string available as an internal object which can be manipulated using the powerful string operations which SETL provides. These facilities make it possible to program an arbitrarily complex 'pretty rint' function in SETL. Such a procedure can indent nested sets and tuples nicely, can sort their elements to make searching easier, etc. Utilities of this kind are well worth developing when large objects need to be printed and inspected repeatedly; in such cases, it is particularly important to sort the output.

To read input from the standard input file (or, for interactive runs, from the terminal) the READ statement is used. This has the form

(1) READ(Rhs1,Rhs2,...,Rhsk),

where each of Rhs1,...,Rhsk is (either) a simple variable (or a more complex expression of the kind which could legally appear on the left-hand side of an assignment statement: see Section 2.11). The statement (1) reads in a sequence of SETL values from the standard input file and makes them the values of Rhs1,Rhs2,...,Rhsk respectively. For example, if the next three items in the input file are

```
{1  2  3}
'HELLO THERE'
[{1},2,A],
```

then the command

```
READ(x,y,z)
```

will give x,y, and z the respective values {1,2,3}, 'HELLO THERE', and [{1},2,'A']. This example illustrates several of the following rules governing the READ primitive.

(i) Successive items in a bracketed SETL value to be read can be separated either by commas or by blanks. For example, to read in the set {1,2,3} we can write its external representation either as

```
{1,2,3}
```

or as

```
{1  2, 3}
```

or as

```
{1  2  3}
```

etc.

(ii) Unbracketed items separated by blanks will be read in by successive READ statements even if they all appear on a single line. None of them will be bypassed, and reading will advance from one line to the next only when more input data is needed to complete the line being read. For example if the first three lines of the input file are

```
      1  2  3  4  5  {6  7
            8  9
  10},
```

then the commands

```
READ(x,y,z);
READ(u,v);
READ(w);
```

will give the variables x through z the same values that they would be given by the following assignments:

```
x := 1; y := 2; z := 3; u := 4; v := 5; w := {6,7,8,9,10};
```

(iii) When read in, valid identifiers, i.e. unbroken strings of letters and numbers starting with a letter, will be read as strings even if they are not enclosed in quote marks. For example, if the input file contains

```
[A  BB  C123],
```

then the command READ(x) will have the same effect as the assignment

```
x := ['A','BB','C123'];
```

(iv) Other items, namely the Boolean values TRUE and FALSE and the undefined atom OM can be read in if they are written in the form in which they would be printed by a PRINT command. In particular, TRUE and FALSE can be read in if they are represented as #T and #F in the input file, and OM can be read in if it is represented as * in the intput file. For example, if the input file contains [*, *, _ T, _ F, *, *] the command READ(x) will give x the same value that the assignment

```
x := [*, *, TRUE, FALSE];
```

would give it. These rules imply that the READ and PRINT operations are almost inverses of each other, i.e. that a file of data written by PRINT can almost be read back in using READ. Unfortunately, this is not quite the case (however, this perfect inverse relationship does hold for SETL 'binary' input/output primitives, namely GETB and PUTB; see Section 8.1 below). For example, if the string 'Hello there' is written out using PRINT, and then read back in using READ, it will appear as the pair 'Hello' 'there' of successive string items. Moreover, if the string 'Hello!there' is written

out using PRINT then any attempt to read the result will cause an error, since the unquoted character '!' happens to be indigestible to the READ primitive. (Also, the external form of an atom, see Section 5.3 below, is indigestible to the READ primitive. Thus READ and PRINT are only inverses to one another if the value being printed and then read back in contains no quoted strings which are not valid identifiers (and also contains no atoms).

As READ operations are successively executed, an implicit 'read position' pointer moves progressively forward in the standard input file, past one SETL value at a time, until eventually the very end of the input file is reached. Thus the input file behaves like a 'tape' on which successive SETL values are written and from which they can be read. Even when the end of the input file has been reached, the READ operation will continue to execute without any error occurring, but in this case all further values read from the input file will be OM. Therefore the input file behaves exactly as if its actual contents were followed by infinitely many OMs. To detect the actual end of input, one must use another SETL primitive operation, represented by the keyword EOF (end of file). This can be used in expressions just like any other variable, but its value is always FALSE if the last READ operation executed did not encounter the end of the input file which it is reading. Conversely, EOF is TRUE if the end of the input file was reached by the last READ statement executed. The value of the quantity EOF changes as soon as a first attempt is made to read past the end of the input file. For example, if the input file contains just the three items {1} {2} {3}, then the loop

```
(FOR j in [1..4]) READ(x); print(x,EOF); END;
```

will produce the output

```
{1}  #F
{2}  #F
{3}  #F
*    #T
```

It follows that to read all data items present in the input and print them out one wants to use a loop which tests the EOF condition immediately after an item is read, as in the following example:

```
LOOP DO                          $ loop to read and echo all items
                                 $   in the input file
   READ(x);
   IF EOF THEN QUIT; END;
   PRINT(x);
END LOOP;
```

If a bracketed item which is not properly closed and one attempts to read it, then a run-time error will occur. For example, any attempt to read an input file whose last two lines are

```
{1,2,3
'HELLO'
```

or whose last line is

{1,2,3]

is fatal. 3.7.1 Reading data from a terminal.

Interactive programs typically take their input from the user terminal. The rules described above also apply to READ statements that take their data from the terminal screen: a READ statement will read as many items as it needs, spread over several lines if need be. If not enough items were supplied ot it, the READ statement will simply wait until the full input is supplied. To indicate that the input is complete, always enter a Carriage Return following the end of the data.

A terminal is a potentially infinite source of data. How is the program to determine that an end-of-file has been encountered in reading from the terminal ? The answer depends on the operating system on which you are running. Special characters are used to indicate end-of-data, and you should find out the conventions used by the operating system on which your SETL system runs. On DEC systems, the character combination Control-Z marks an end-of-file, so that entering CRTL-Z will make the EOF test true.

3.7.2 Character sets.

The simple READ and PRINT primitives described in this section get input from the standard input file and send output to the standard output file. As explained in Section XXX, if input is to be read from the standard input file the lines of data constituting this file should be supplied following your SETL program (for a batch run) and should be typed in interactively at a terminal (in an interactive run). Other more advanced input-output primitives (described in Section 8.1 below) allow output to be read from and written to other files. These files are made available to your SETL program in a manner which necessarily depends (to a certain extent at least) on the operating system being used. See Section Y for additional details.

As noted in Section Z, not all the computer systems on which SETL runs support the full character set assumed in this book. Where particular characters are missing, they are represented either by single substitute characters which are available, or by pairs of such characters. Obviously, this will affect some details of the output produced and of the input expected by the READ primitive. See Appendix XXX for additional details concerning alternative character representations. Note that the character set which SETL will use can sometimes be controlled by supplying appropriate control card parameters. See Section W for details.

3.8 Exercises

Ex. 1 Write a program which reads a set s of integers and prints out a list, in ascending order, of all the members of s which are prime.

Ex. 2 A set of vectors of length n and a vector x of length n are given. Write code which selects the elements of s which has the largest number of components in common with x.

Ex. 3 Write a program to read a character string, reverse the order of its characters, and print it out.

Ex. 4 Write a program that will scan a string of characters containing parenthesis, square brackets, and set brackets, and determine whether it is properly bracketed. (A string is properly bracketed if each left bracket or parenthesis is matched by a following right bracket or parenthesis of the same kind. For example, {[ ]} is properly bracketed, but {[}] is not.)

Ex. 5 Write a program that reads in successive pairs of strings s,t, of the same length, and determines whether t can be obtained from s by substituting for the characters of s in some single-valued way. For example, 'ipstf' is obtained from 'horse' by the substitution {['h','v'], ['o','p'], ['r','s'], ['s','t'], ['e','f']}, but 'beer' cannot be obtained from 'anna' in this way, since two different characters would have to be substituted for 'a'.

Ex. 6 Write a program which will translate an arbitrary message into Morse code. The Morse codes for all characters of the alphabet and for the commonest punctuation marks are shown in Fig. X. Write a program which will translate Morse code back into English.

Figure : Morse cdes for alphabetic and special characters.

Ex. 7 A publisher produces books both in hard cover and paperback. Any given book can be either long, medium, or short, and can be either elementary or advanced. A short, elementary, paperback book sells for $5. Exactly $2 is added to the price of a book if it is hard cover; medium-length books sell for exactly $1 more, and long books for exactly $3 more, than short books. The price of a book is doubled if it is advanced. Write a small program which will print out all possible categories of books together with their prices.

Ex. 8 Write a program that will read an integer n and print its successive digits separated by spaces, starting with its leftmost digit.

Ex. 9 Write a program which can read an arbitrary integer and print it in English. For example, -143 should print as 'minus one hundred fourty-three'. Can you do the same for French? For German? For Chinese?

Ex. 10 Write a program to read in three points x,y,z, each represented by a pair of real numbers. Determine whether these three points:

(a) all lie along a line; (b) form the corners of an isoceles or equilateral triangle; (c) form the corners of a right triangle.

Print out an appropriate message in each case.

Ex. 11 Write a program which will read in a sequence of lines, each containing someone's name, first name first, and print out an alphabetised list of these names, in alphabetic order of last names. Repeat this exercise, but this time print the alphabetised list with last names first.

Ex. 12 Making use of a map from family names into their probable ethnic origins, write a program which reads a list of names and attempts to guess the ethnic origins of their bearers. Your program should also make use of facts like the following to increase its coverage: names beginning with 'Me' are probably Irish, with 'Mac' probably Scottish; names ending in 'ski' are probably Polish, in 'ian' probably Armenian, in 'wetz' probably East-European Jewish, in 'ini' probably Italian, etc. How well does your program guess the family origins of your classmates?

Modify this program so that it uses first names to guess sex. Note here that names ending in 'a' are probably female, etc.

Ex. 13 A college collects statistics on the members of its entering freshman class. The basic data for each student is a line in a data file, consisting of the following items, in sequence, seperated by blanks:

student's last name, first name, age(in years), sex(M or F),

maritial status (0=single, 1=married, 2=divorced or separated)

Write a program to print out the following information:

(i) Percent under 21 years old
(ii) Percent over 21 years old
(iii) Percent over 30 years old
(iv) Percent male and female
(v) Percent of males single, married, and divorced or separated.
(vi) Percent of females single, married, divorced, or separated.

Ex. 14 An automobile sales agency employs 25 salespersons. Sales records are kept on cards, each card holding the following information, separated by blanks:

(a) Salesman's last name
(b) Make of car sold

(c) Amount of sale
(d) Net amount of sale (i.e., total amount minus discount allowed for trade-in.) Write a program which will read a monthly file of such cards and print out the commission due to each salesperson. The rules determining commissions are as follows:

(i) Standard commission is 5% on the first $20,000 of net sales, 6% on the next $10,000 of net sales, and 7% on all sales over $30,000.

(ii) Indvidual sales totaling more than $10,000 earn a 1% bonus.

(iii) Sales on which less than $500 trade-in is allowed earn a bonus of one half of 1%.

Ex. 15 A factory's payroll is prepared from a set of daily time cards and a mapping f giving the hourly wage rate for each employee. Each time card contains an employee's social security number followed by the number of hours worked on a particular day. The mapping f sends each employee social security number into the employee's name, hourly wage rate and tax withholding rate. Total pay is number of hours worked, times hourly base rate, times (1-r), where r is the tax witholding rate; however, all hours in excess of 40 are paid at a time -and-a-half rate. Write a program to read a file representing a week's payroll records, and print out a payroll showing employee name, social security number, total pay, tax withheld, and net pay.

Ex. 16 Suppose that the daily time cards of Exercise 18 are grouped into batches separated by cards which contain only the single digit 0, with the Monday batch coming first, Tuesday next, etc., and that work performed on weekends is paid at a double-time rate. Modify the program of Exercise 18 to handle this rule also.

Ex 17 In bowling, a complete game consists of ten frames. Either one or two balls is rolled in a frame. If all ten pins are knocked down by the first ball rolled in a frame (this event is called a 'strike') the score for the frame is 10, plus the number of pins knocked down by the next two pins rolled. If all ten pins are knocked down by the two balls rolled in a frame (called a 'spare), the score for the frame is 10, plus the number of pins knocked down by the next ball rolled. Otherwise the score for the frame is the number of pins knocked down by the two balls rolled in the frame. If a spare is rolled in the tenth frame, then you are allowed an extra ball; if a spare is rolled in the tenth frame, then you are allowed two extra balls, so can earn up to 20 more points.

Write a program which will read a tuple representing the number of pins knocked down by each ball rolled during a game and print out the corresponding score.

Ex. 18 Explain how the conditional statement

```
IF C1 THEN block_of_statements_1
ELSEIF C2 THEN block_of_statements_2
ELSEIF...
ELSE block_of_statements_n
```

can be re-expressed using IF-statements of the simple form

```
IF C THEN GOTO label_j
```

but no other conditional statements.

$

HAPTER 4

FUNCTIONS AND PROCEDURES

A function in SETL is a computational process which has been given a name and which, using one or more data items passed to it, will compute and deliver a value. Most of the built-in SETL operators, for example MAX, which returns the maximum of two values x and y, and COS, which returns the cosine of a floating point number x passed to it, are functions in this sense. However, since no finite collection will ever exhaust the whole catalog of functions that a programmer may want to use, it is important to have a way of defining, and then using, as many additional operations as are helpful.

Chapter Table of Contents

4.1 Writing and using functions

To make the above point more convincing, we can consider a simple example. Suppose that some numerical quantity associated with a relatively standard product, for example the weight of eggs coming from a chicken farm, is measured daily, thus producing batches of measurements, each of which can

be thought of as a set of numbers, e.g.

(1) {2.7, 2.85, 1.90, ...,1.85}.

Suppose that in order to exert some sort of quality control, various statistical properties are to be reported for each such batch, and that these statistics are to include the weights of the three largest and the three smallest eggs in the batch.

To make this calculation easily, it would be convenient to use a pre-programmedfunction to which a set s like (1) can be passed, and which would then produce a tuple t

(2) [1.86, 1.90, ..., 2.7, 2.85]

in which all this members of s are arranged in increasing order. Since the function would simply sort the members of s, it might appropriately be called -sort-. We would like to be able to produce the ordered tuple t from the set (1) simply by writing

(3) t:=sort(s).

Note that if this can be done, then to print the three largest and three smallest measurements we have only to write

```
print('three smallest measurements are:',t(1),t(2),t(3));
print('three largest measurements are:',t(_ t),t(_ t-1),t(_ t-2));
```

Of course, sorting the set s is not hard, and can be done by the simple method explained in Section 3.5.1, which is to say using the code

(4)
```
t:=[];

(WHILE s/={ })
   t WITH:=(x:=min/s);
   s LESS:=x;
END WHILE;
```

However, what we want is to package the code (4), giving it the name -sort- and invoking it by the name. By doing this we make it possible to get the effect of the code (4), without having to concern ourselves with its inner workings, simply by writing (3). To 'package' bits of code in this way becomes absolutely essential when one is constructing large programs (say a few hundred lines or more). Programs of such sizes can only be built succesfully if they are organized hierarchically into a modular collection of subprocedures; typically such a collection will include both high-level functions which simply make use of facilities provided by lower level functions, and low level procedures, like the "sort" which we have been discussing, which encapsulate general, useful primitive operations. SETL does provide a facility for defining as many new functions as you need, and we now proceed to explain how this is done.

To 'package' or 'encapsulate' the code (4), all we need to do is to enclose it between procedure 'header' and 'trailer' lines, and add a RETURN statement. This gives

```
(5)     PROCEDURE sort(s);

        t:=[];

        (WHILE s/={ })
           t WITH:=(x:=MIN/s);
           s LESS:=x;
        END WHILE;

        RETURN t;

        END PROCEDURE sort;
```

In (5) the function header line is

```
(5a)    PROCEDURE sort(s);
```

This line, introduced by the special keyword PROCEDURE (which can also be abbreviated as PROC), opens the procedure (5), gives it a name (in this case, the name -sort-), and also names its formal parameters, (sometimes simply called parameters) i.e. the values which will be passed to the function whenever it is used (as in (3) above), and from which the function will calculate the value which it returns. (In (5), the value returned is t and there is only one formal parameter, namely -s-.) The concluding trailer line

```
(5b)          END PROCEDURE sort;
```

marks the end of the function. (And, in a large program making use of many functions, would ordinarily be followed by the header line of another function.)

The command

```
(5c)           RETURN t;
```

which appears in (5) indicates the point at which the function has finished calculating the value which it is to produce, and also defines the value that the function will return.

To 'use' or 'invoke' the function -sort- defined by (5), we have only to write sort(e), where e can be any expression. This automatically calculates and makes available the value returned by the function (5). For example, if we write

```
(5d)          print(sort({5, 1, 2, 7, 0}))
```

the result will be

```
              [0 1 2 5 7]
```

The expression e occuring in such a 'use' or 'invocation' sort(e) of the PROCEDURE "sort" is called the actual argument, or supplied argument of the invocation. Whenever evaluation of a function invocation like (5d) begins,

the value of the actual argument (or arguments) appearing in it is transmitted to the procedure invoked, and becomes the initial value of the procedure's formal parameter (or parameters).

To explain more of the details involved in the use of SETL functions, we will consider a simple invocation of such a function, namely

(6) x := sort({5, 1, 2, 7, 0});

As with all assignment statements, execution of (6) begins with evaluation of its right-hand side. Since -sort- is the name of a function, evaluation of the function invocation appearing on the right-hand side of the assignment (6) involves the following steps:

(i) The current value of the actual argument {5, 1, 2, 7, 0} of the function invocation is assigned as the initial value of the formal parameter variable s appearing in the procedure 'code' or 'body' (5).

(ii) Execution of the procedure (5) begins: the statements appearing in the body of this procedure are executed in the ordinary way. Wherever the formal parameter appear, in the body of the procedure, the value of the actual parameter passed to the function is used.

(iii) As soon as any RETURN statement is encountered, control is passed back from the procedure (5) to the instruction immediately following the invocation (6). Just before this happens, the expression following the keyword RETURN is evaluated and becomes the value which the function (5) yields. (E.g., becomes the value of the variable -x- in (6).)

The 'detour and return' action typical of function invocations is shown schematically in the following diagram:

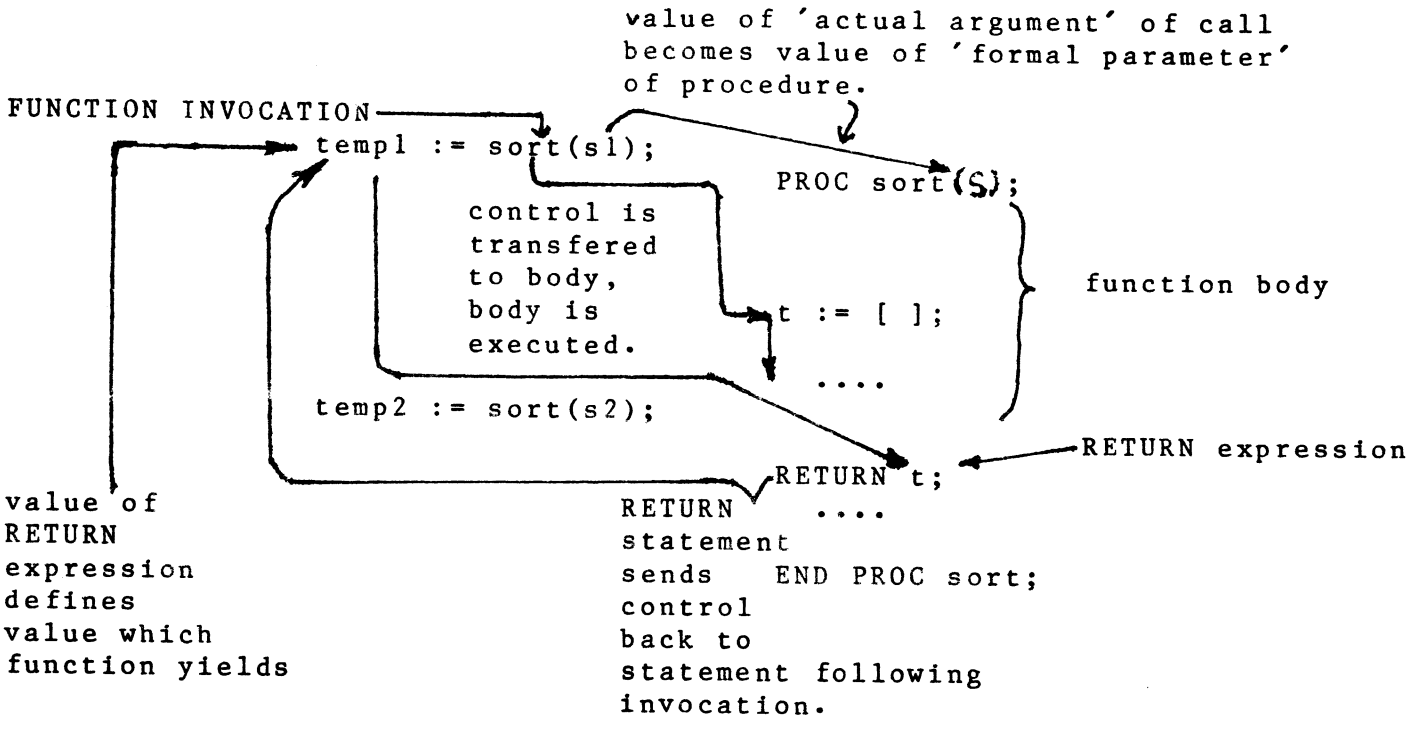


Figure 4.1 Detour and return in function invocations

The following analogy should help to clarify the important distinction between the 'formal parameters' and the 'actual arguments' of a procedure. The formal parameters of a procedure can be compared to the item names which occur in a cook-book recipe. For example, a recipe may say 'break an egg into half a cup of flour and stir for 24 hours or until the mixture becomes firm'. The names 'egg' and 'flour' appearing in such a recipe are 'formal names' which stand for all the actual eggs and actual half cups of flour which will be needed when the recipe is actually used. As in the case of a function, new actual items, i.e. a different egg and a different half cup of flour, must be supplied each time the recipe is used, even though the formal names 'egg' and 'flour' appearing in the recipe remain the same. Continuing this analogy, the text of the recipe can be compared to the body of a function, which will yield something (e.g. a cake) when 'actual' ingredients matching the 'formal' ingredients to which it refers are passed to it.

It is instructive to consider a somewhat more complicated example, namely

```
(6b)            x:=sort(s1) + sort(s2) ;
```

Suppose, for example, that s1 and s2 happen to have the values {3,1,0} and {-3,-1,0} respectively when (6b) is executed. Then evaluation of sort(s1) will produce the value [0,1,3] and evaluation of sort(s2) will produce the value [-3,-1,0], so that after (6b) is executed the variable x will have the value [0,1,3,-3,-1,0].

The way this happens is as follows. As with all assignment statements, execution of (6b) begins with evaluation of its right-hand side, i.e. sort(s1) + sort(s2). This is an expression, and is evaluated by first evaluating its two subexpressions sort(s1) and sort(s2) and then combining the two resulting values using the '+' operator. As always,the leftmost of these two subexpressions, namely sort(s1), is evaluated first.

These details are more accurately and fully represented if we break up the evaluation of (6b) into smaller steps, as follows. This is considerably closer than the 'source text' (6b) to the so-called 'internal text'or 'directly executable code' which the SETL system actually uses:

```
(7)             temp1 := sort(s1) ;
                temp2 := sort(s2) ;
                x := temp1 + temp2 ;
```

(The additional variables temp1 and temp2 which appear in this code are so-called 'compiler temporaries'. They are automatically generated by the SETL compiler to store necessary intermediate values and are not in any way directly accessible to the ordinary SETL user.) As you can see, (7) involves two succesive invocations of -sort-, followed by a use of the '+' operator to combine the two results produced.

The following important rules govern the use of functions.

a) The formal parameters that appear in the procedure heading must be valid identifiers, that is to say they are variable names; furthermore no two parameters can be the same. For example, both

(8a) PROCEDURE p1(s*t) ;

and

(8b) PROCEDURE p2(s,t,s) ;

are illegal; (8a) because the parameter s*t is not a simple variable, and (8b) because the first and the third formal parameters of p2 are identical. On the other hand, any actual argument of a function invocation can be an (arbitrarily complicated) expression, and actual arguments can be repeated. For example,

(9a) x := sort({x IN ss|x>0}) ;

is legal if ss is a set (and if ss were {-10,20,-20,15,10} , would give x the value [10,15,20]). Similarly, if dot_prod(x,y) is a function which calculates and returns the dot-product of the two tuples x and y, then

(9b) a := dot_prod(u,u)

is legal (and will put the sum of the squared components of the tuple u into a).

(b) Each invocation of a function must have exactly as many actual arguments as the function has formal parameters. (However, it is possible to define functions and procedures for which this rule is relaxed, see Section 4.7.1). When a function is invoked, the value of its first (resp. second, third, etc.) argument becomes the value of its first (resp, second, third, etc) formal parameter. For example, if the function whose header line is

```
PROCEDURE intermingle(a,b,c) ;
```

is invoked by

```
x := intermingle({x IN s | x>0}, {y IN s2|y<0}, {x IN s|x>0}) ;
```

then a and c initially get the value {x IN s | x>0}, while the value {y IN s2 | y<0} is transmitted to b.

(c) The body of a function can contain any number of RETURN statements, and often will contain more than one. The following code, which simply calculates and returns the maximum of two quantities, exemplifies this remark:

```
PROCEDURE my_very_own_max_function(x,y);

IF x > y THEN
    RETURN x ;
```

```
    ELSE
        RETURN y ;
    END IF ;

    END PROCEDURE my_very_own_max_function ;
```

Generally speaking, a RETURN statement should be inserted at whatever point or points in the body of a procedure at which the value which the procedure is supposed to yield has been calculated; an expression yielding this value must follow the keyword RETURN. Since the point at which a function has calculated its intended result will often depend on the actual value of the arguments passed to it, it is often appropriate to write RETURN statements at several points in a function's body. In this case, whichever RETURN statement that is executed first will end the execution of the function (and return control to the routine which invoked the function). If no RETURN statement is encountered, execution of the function will terminate when and if its trailer line END PROC is reached, and in this case the undefined value OM will be returned.

Note that the keyword RETURN can be followed by an arbitrary expression. This expression is calculated immediately before control is switched back from the function to the point at which it was invoked. If such a 'return expression' is at all complex, the whole body of the function may simply consist of a single RETURN statement and nothing else, as in

```
PROCEDURE positive_elements_in(s);
      $ returns the set of positive elements of s

RETURN {x IN s | x > 0};

END PROCEDURE positive_elements_in ;
```

(d) Functions can invoke other functions (including themelves) without restriction. When control is transferred to a function f which in turn invokes a function g, execution will proceed within the body of f until an invocation of g is encountered, at which point execution of f will be suspended and execution of g will begin. Thereafter, g will execute until a RETURN statement is encountered within g, at which point g will terminate, sending control, and possibly a value, back to f. Subsequently, when a RETURN statement is encountered in f, f will itself be terminated, sending control (and a value) back to the procedure from which f was invoked. This will lead to patterns of control transfer like that shown in the following figure.

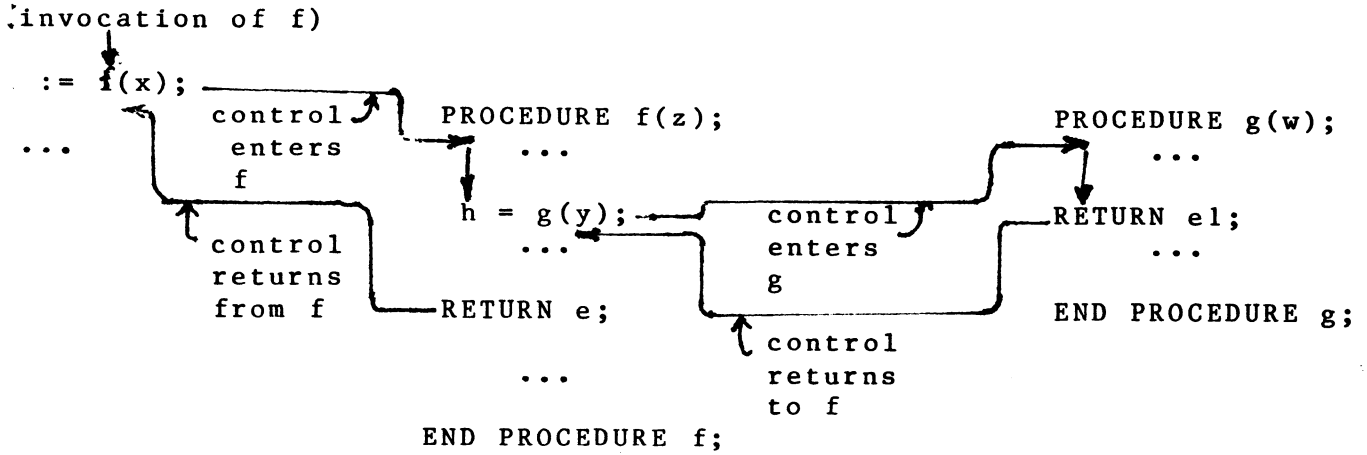


Figure 4.2 Patterns of control transfer in multiple function calls.

(e) Function invocations are themselves expressions, and can be used freely as parts of more complex expressions. For example, if -sort- is a function which returns the elements of a set s in sorted order as a tuple, and -sum_square- is a function which returns the sum of the squares of the three first elements of a tuple, then we can write

```
print(sum_square(sort(s)));
```

to display the sum of the three smallest elements of s.

4.1.1 Some simple sorting procedures.

One of the simplest sorting procedures is the so-called 'bubble-sort' method, which simply stated operates as follows : as long as there are two adajacent elements that are out of order in the sequence, permute them. This is not a very efficient sorting method (and in the form presented below it is even more inefficient than the standard bubble sort) but it is one of the simplest to state and program. The input to the procedure is a tuple and the output is another tuple, whose elements are in increasing order. Note that the code that follows applies equally well to a tuple of integers, a tuple of floating point numbers, or a tuple of strings: in all three cases the comparison operator '>' defines the desired ordering.

```
PROC sort(t);   $ sorts a tuple by the bubble-sort method

(WHILE EXISTS i IN [1..#t-1] | t(i) > t(i+1))
          [t(i), t(i+1)] := [t(i+1), t(i)] ;
END WHILE ;

RETURN t ;

END PROC sort ;
```

The attentive reader may wonder whether it is dangerous for this function to modify its own parameter t. In fact, doing so causes no problems; but the rule guaranteeing this will only be stated in Section 4.5. (This same remark also applies to several of the functions presented later in this section.)

As we mentioned, the function shown just above can be used to sort any tuple of integers, of reals, or of strings. For example, if we write

```
print(sort({'Joe', 'Ralph', 'Albert', 'Cynthia', 'Robert', 'Alfredo'}))
```

the result would be

```
[Albert Alfredo Cynthia Joe Ralph Robert]
```

More complex sorting routines than that shown above are often needed. One reason for this is that sorting is often used to arrange more complex 'records' into an order determined by some common 'subfield' of the records. In SETL, such records are typically represented as tuples. Suppose, for

example, that a group of students have taken a course in which their grades on a series of homework exercises and examinations have been collected, producing a set of tuples having the following form:

```
records:={['Gonzalez,Aldo', 80,87,OM,73,90,..],
          ['Whyburn, Linda', 82, 89, 85, 91, 90, 65,..],
          ['Luciano, Luigi', 80, 81, 75, 79, OM, 70,..],
          ...}
```

Grades are assumed to be represented by integers, and missed exercises or examinations by occurences of OM. One might then want to list these records into various orders, e.g.

(a) Alphabetic order of student names
(b) Order of grade averages, with largest first
(c) Order of grades on mid-term examination, largest first
(d) Order of number of exercises not handed in, largest first, etc.

To make it easy to sort these records according to any of their fields, we modify our original sorting procedure, so that it takes two arguments:

i) The tuple of records to be sorted.

ii) The record component by which the records must be sorted.

This leads to the following procedure:

```
PROCEDURE sort1(t, pos) ;

$ T is a tuple of records (tuples) to be sorted.
$ pos is the index of the component in each record, along which
$ the records are to be sorted in increasing order.

(WHILE EXISTS i in [1..#t-1] |
                        t(i)(pos) > t(i+1)(pos) )
    [t(i), t(i+1)] := [t(i+1), t(i)] ;
END WHILE ;

RETURN t ;

END ;
```

Using this function, we can print the class records in alphabetical order simply by writing

```
(FOR x in sort1(records, 1))
     print(x) ;
END ;
```

Suppose now that we want to list these records in order of decreasing midterm grades, with students that have missed the mid-term coming last. If the mid-term is the 11-th entry in the record, we may be tempted to sort the records along that component (in increasing order) and then list them in reverse. The attentive reader will notice that sort1 will not work very

well in the presence of missing grades: recall the convention that a missed test is marked as OM in the record. The comparison (OM > x) where x is same value, is not meaningful, and in fact the SETL system will stop any program at the point at which such a comparison is attempted. As a final modification to our sorting procedure, let us replace the comparison that drives the WHILE loop, so that a value of OM is regarded as smaller than any existing grade. Using the 'is_undefined' operator, we simply replace t(i)(pos) by t(i)(pos) ? (-1). Our improved sorting routine now reads:

```
PROCEDURE sort2(t, pos) ;

$ T is a tuple of records, some of whose components may be OM.
$ pos is the index of the record component along which the records
$ are to be sorted in increasing order.

(WHILE EXISTS i in [1..#t -1] |
                  t(i)(pos) ? (-1)  >  t(i+1)(pos)  ?  (-1))
    [t(i), t(i+1)] := [t(i+1), t(i)] ;
END WHILE ;

RETURN t ;
END ;
```

With this modification, we can print the desired ordering by midterm grades with the following code (recall that the name is the first component of the record, the midterm grade is the 11-th component of the record, and that this grade may be undefined):

```
ordered := sort2(records, 11) ;
(FOR i in [ ordered,  ordered -1..1])
    print(ordered(i)(1),  ordered(i)(11) ?  '** absent **') ;
END FOR ;
```

Other plausible applications of this same kind appear in exercises XXX-YYY.

The 'main program block'.

A program which makes use of sub-PROCEDUREs must of course include commands which invoke these subprocedures. As we have explained, the first function invoked can invoke any or all of the other functions, but at least one instruction not belonging to any PROCEDURE is needed to trigger this first invocation. In a program including one or more PROCEDUREs, the initial, 'directly executed' portion of the program, i.e. everything not included in any sub-PROCEDURE, is called the main block of the program, or the main program for short. This block of instructions has exactly the form of a PROGRAM body, as described in Chapters II and III, and it must precede all PROCEDURES. The main program and all the procedures which follow it must be prefixed by a PROGRAM header line of the usual form, and a corresponding trailer line starting with the keyword END must follow the last procedure.

For example, a complete program consisting of the -sort- function shown above and the two fragments of code which invoke it would have the following overall structure:

```
PROGRAM print_grade_info;    $ program to print student grade records

    read(records);           $ acquire the basic data
    print('student records in alphabetical order') ;
    print('-----------------------------------') ;

    (FOR x IN sort(records, 1))
        print(x) ;
    END ;
    print('students and mid-term grades, in decreasing grade order');
    print('-------------------------------------------------------') ;
    ordered := sort(records, 11) ;
    (FOR i in [#ordered, #ordered-1..1])
         print(ordered(i)(1),  ordered(i)(11) ? ' ** absent ** ') ;
    END FOR;

    PROC sort(t, pos);
    $ t is a tuple of records.
    $ pos is the position of the record component according to which
    $ the records are to be sorted in increasing order.

    (WHILE EXISTS i IN [1..#t-1] |
                          t(i)(pos) ? (-1) > t(i+1)(pos) ? (-1) )
         [t(i),t(i+1)]:=[t(i+1),t(i)];
    END WHILE;

    RETURN t ;

    END PROC sort;

END PROGRAM print_grade_info;
```

Execution of such a program begins at the first statement of its main program block and ends as soon as the last statement of its main program block has been executed (or when a STOP statement is encountered; see Section XXX).

4.1.2 A character-conversion procedure

We continue to present an illustrative series of functions. Our next function takes a string and returns a similar string in which all lower-case alphabetic characters have been changed into the corresponding upper-case characters. Blanks and punctuation marks are not affected.

```
PROC capitalize(s); $ capitalizes the string s and returns
                    $ the result. Non-alphabetic characters are left
                    $ alone

small_letters := 'abcdefghijklmnopqrstuvwxyz';
big_letters   := 'ABCDEFGHIJKLMNOPQRSTUVWXYZ';

capital_of  := {[l,big_letters(i)]: l=small_letters(i)};          $
maps each small letter into the corresponding capital.

RETURN +/[capital_of(let) ? let : let = s(i)] ;
```

```
        $ Note that the map capital_of is defined over
        $ alphabetic characters only.  Non-alphabetic
        $ characters, such as punctuation marks,
        $ are not converted but left as they are.
        $ This is the purpose of the '? let' expression.
    END PROC capitalize;
```

A function can have any number of parameters. Occasionally it is even appropriate to write functions which have no parameters. For example, we may want to use a function which reads an input string, uses the -capitalize- procedure to capitalize this input, and returns the capitalized result. This function can be written as follows:

```
    PROC next_line;  $ procedure to read and capitalize a line

    read(x);           $ read a quoted string

    RETURN IF x=OM THEN OM ELSE capitalize(x) END;
                       $ return its capitalized form.
    END PROC next_line;
```

To invoke a parameterless procedure of this sort, one writes its name, followed by an empty parameter list. For example, to invoke the -next_line- function and print the capitalized string which it returns, we would write

```
    print(next_line( ));
```

Note that the empty parameter list, i.e. the '( )' following the name of parameterless function -next_line-, is obligatory.

(c) A package of procedures for manipulating polynomials

As a further illustration of the use of functions, we give a set of procedures for adding, subtracting, multiplying, and dividing in polynomials a single variable polynomials with real coefficients . Such polynomials are ordinarily printed in a standard algebraic form like

```
    3.1*x**2 + 7.7*x+4.5,
```

but in the procedures which follow we will assume that a polynomial is represented internally by a SETL map which sends the exponent of each term of the polynomial into the coefficient of that term. For example, the polynomial shown above would be represented internally by the map

```
    {[2, 3.1], [1, 7.7], [0, 4.5]}.
```

As in algebra, we simply omit terms whose coefficients are zero.

To add (resp. subtract) two polynomials, we simply add (resp. subtract) the coefficients of corresponding terms. Hence functions which calculate polynomial sums and differences can simply be written as follows:

```
PROC sum(p1,p2);     $forms the sum of two polynomials.

RETURN {[e,c]: c1=p1(e) | (c := c1+(p2(e) ? 0.0)) /= 0.0}
    + {[e,c2]: c2=p2(e) | p1(e)=OM};

END PROC sum;

PROC diff(p1,p2);     $forms the difference of two polynomials

RETURN {[e,c]: c1=p1(e)  | (c := c1-(p2(e)?0.0)) /= 0.0}
    + {[e,-c2]: c2=p2(e) | p1(e)=OM} ;

END PROC diff ;
```

To multiply two polynomials, we can simply multiply all pairs of their individual terms, and then group together and sum all resulting terms having identical exponents. Finally, we eliminate terms which turn out to have zero coefficients. This is simply

```
PROC  prod(p1,p2) ;     $ forms the product of two polynomials

p:={[e1+e2, c1*c2]:  c1=p1(e1), c2=p2(e2)};

RETURN {[e,c]: all_coeffs=p{e} | (c := +/all_coeffs) /= 0}

END PROC prod ;
```

Next, we show how to divide a polynomial p1 by a polynomial p2. Let c1*x**j1 be the leading term of p1 i.e. the term having largest exponent, and let c2*x**j2 be the leading term of p2. Then we subtract c1/c2*x**(j1-j2) times p2 from p1, to eliminate the leading term of p1, and so on repeatedly until all terms of p1 with exponents larger than j2 have been eliminated. The collection of all terms by which p2 is multiplied constitute the terms of the quotient.

```
PROC div(p1,p2);                  $ forms the quotient polynomial p1/p2

if p2={ } THEN RETURN OM; END;  $ this is the case p2=0.

e1 := MAX/[e:c=p1(e)];            $ largest exponent of p1
e2 := MAX/[e:c=p2(e)];            $ largest exponent of p2

qcoeff:={ };                      $ start with an empty quotient

(FOR j in [e1-e2,e1-e2-1..0] | p1(e2+j)/=0.0)
    qcoeff(j) := p1(e2+j)/p2(e2);
    p1 := diff(p1, {[e+j, qcoeff(j)*c] : c=p2(e)} ) ;
END FOR;

RETURN qcoeff;     $     return the map representing the quotient.

END PROC div;
```

We note that techniques for manipulating polynomials by computer have been studied very intensively, and that much more efficient methods than those used in these simple illustrative procedures are known. See Knuth, The Art of Computer Programming, Vol.2, for an account of these developments, which go beyond the scope of the present book.

4.2 Name scopes; local and global variable names. The VAR declaration.

In writing a long program, which can involve hundreds of procedures, it is irritating, as well a highly error-inducing, to have to remember which variables had been used for which purposes through the whole of a long text. To see this, consider the plausible case of a function invocation imbedded in a WHILE loop like

```
(1)     i := 0 ; j := 0 ;
        (WHILE (i+j) < f(j))...
```

and suppose that f is an invocation of a function whose body is found somewhere else in a long program text. It is entirely plausible that, unknown to the author of the code (1), the body of the function f should make use of the convenient variable name i, e.g. in a loop like

```
(2)     (FORALL i IN [1..#t])...
```

But then, if the i appearing in (1) and the i appearing in (2) were regarded as representing the same variable, the function invocation f(j) which occurs in the WHILE loop could change the value of i in ways not at all hinted at by the outward form of the code(1). Were this the case, a programer wishing to write a loop like (1) would first have to examine the body of the function f, note all the variables which it used, and carefully avoid all unplanned use of similarly named variables. This would introduce many higly undesirable interactions between widely separated parts of a lengthy program, and make large programs harder to write.

To avoid these very undesirable effects, most programming languages make use of rules which restrict the scope of names. The SETL rule is as follows. In the absence of explicit declarations, variables retain their meaning only within a single procedure (or main program). This implies that ordinarily a variable i appearing in one procedure and a variable i appearing in another procedure are treated as distinct. In effect, the SETL compiler applies the following renaming procedure the program text which it processes:

(a) The main program which begins the program text is numbered zero, and the procedures which follow this main program are numbered 1,2,.. in their order of occurence.

(b) Every variable name xxx used in the n-th procedure, including the names of its formal parameters, is implicitly changed to xxx_n.

As an example, consider the program

```
PROGRAM example;
```

```
    x := {3,0,1,2};
    print(squares(sort({i in x:i > 0})));

    PROC sort(i);    $ sorts by the 'quicksort' method

    RETURN IF (x:=ARB i)=OM THEN [ ]
       ELSE sort({e in i | e < x}) + [x]
          + sort({e IN i | e > x})  END ;

    END PROC sort;

    PROC squares(x);  $ forms and returns the tuple of squares of the
                      $ components of the tuple x

    RETURN [e*e : e=x(i)];

    END PROC squares;

END PROGRAM ;
```

Given the above program as input, the SETL compiler will implicity apply the renaming rules (a),(b), and therefore it will really see the following renamed variant:

```
PROGRAM example;

    x_0:={3,0,1,2};     $ main program
    print(squares(sort({i_0 in x_0:i_0 > 0})));

    PROC sort(i_1);    $ subfunction number 1

    RETURN IF (x_1:=ARB i_1)=OM THEN [ ]
       ELSE sort({e_1 IN i_1 | e_1 < x_1}) + [x_1]
          + sort({e_1 IN x_1 | e_1 > x_1})  END ;

    END PROC sort;

    PROC squares(x_2);     $ subfunction number 2

    RETURN [e_2*e_2 : e_2=x_2(i_2)];

    END PROC squares;

END PROGRAM ;
```

As stated above, rule (b) serves to isolate variables of the same name from each other if they are used in different procedures. Variables used in this way are said to be local to the procedures in which they appear. This is generally what we want. However, in some cases, we do want a variable used in several procedures to refer to the same object in all of them. For example, one or more 'major' data objects may be used by all the functions in a related group of functions and in this case it can be convenient to allow all the functions to refer to these objects directly. To see this, consider the case of a group of functions written as part of an inquiry system to be used by the executives of a bank. This might involve many

functions, for example

```
PROC payments(customer_name);  $ returns a given customer's payment
    ...                         $ record

PROC tel_no(customer_name);    $ returns a given customer's
                                $ telephone number
    ...

PROC overdue(ndays);    $ returns set of a customers whose payments
                        $ are more than ndays overdue
...etc.,etc.
```

It should be plain that all these routines will have to make use of one or more 'master files'. (When represented in SETL, these 'files' are likely to be sets of tuples representing records, maps sending customer names, or perhaps customer identifiers such as social-security or account numbers, into associated records, etc.) Instead of insisting that these 'master files' be passed as arguments to all the functions which need to use them, it is more reasonable to make them available directly to every function, giving them easily recognizable variable names such as -master_customer_file-. To make this possible, SETL provides a special form of statement, called the VAR declaration. By writing

```
VAR master_customer_file;
```

at the very start of the overall PROGRAM in which the functions listed above appears, we make -master_customer_file- a <u>global</u> variable which designates the same object in all the functions which reference this variable. The required layout of a program using one or more global variables is shown in the following example:

```
PROGRAM banking_system;   $ header line for overall program

VAR master_customer_file; $declaration of global variable
   (additional global variable declarations come here)

   (body of 'main' program of banking_system comes here)

PROC payments (customer_name);  $first subfunction
    ...
END PROC payments;

PROC tel_no;       $ second subfunction
    ...
END PROC tel_no;

PROC overdue;       $third subfunction
    ...
END PROC overdue;

  (additional functions and subprocedures come here)

END PROGRAM banking_system;
```

The statement

```
VAR master_customer_file;
```

appearing first in this example is called a declaration rather than an executable statement because it serves to modify the meaning of other statements rather than to trigger any particular calculation.

The general form of a VAR declaration is

```
VAR  x1,x2,...,xn;
```

i.e. it consists of the keyword VAR followed by a comma-separated list of distinct variable identifiers.

Such declarations can appear in one of two positions:

(a) In a PROGRAM, before the program's first executable statement. Variable identifiers appearing in such a declaration are defined to be global variables directly accessible to each following PROCEDURE in the program. A VAR declaration appearing in this position is called a global VAR declaration.

(b) In a PROCEDURE within a program, before the procedure's first executable statement. A VAR declaration appearing in this postion is called a local VAR declaration. Variable identifiers appearing in such a declaration are defined to be local variables accessible only within the procedure. Since variable names not appearing in any VAR declaration are in any case local to the procedures in which they appear, VAR declarations appearing in this position often serve only to document the way in which a procedure uses its variables. However, if the procedure is recursive (see Section 4.4), VAR declarations appearing in it have a more significant effect, which will be described more fully in Section XXX below.

Any number of VAR declarations may appear either at the start of a PROGRAM or within a PROCEDURE, but all such declarations must precede the first executable statement of the PROGRAM or PROCEDURE in which they appear. No variable should appear twice in VAR declarations (either global VAR declarations or declarations within a single procedure), nor is it legal for any procedure parameter name to appear in a global VAR declaration. See section 9.1 for the rules which apply to VAR declarations appearing in DIRECTORYs, MODULEs, and LIBRARYs within large, complex SETL programs; see Section YYY for an account of the modified VAR declaration used to declare backtracked variables.

A global variable retains its value between invocations of the procedures which use it. (The same remark applies to a variable appearing in a VAR declaration within a procedure).

To sum up, there are two ways in which values can be communicated between seperate PROCEDUREs.

(i) By being passed as parameters or RETURNed as function values.

(ii) By direct global communication, i.e. by being the values of variables which have been declared to be global and hence are accessible to more than one procedure.

Method (ii) is powerful, but potentially undisciplined, since it allows functions to influence each other in ways that their invocations hide. It is therefore good programming practice to avoid using more than a very few declared global variables. Generally speaking, variables should be made global only if

(A) They represent 'major' data objects accessed by most of a PROGRAM's functions, and their usage is subject to clearly understood rules of style which pervade the entire program.

(B) They represent 'flags' or other conditions which many procedures need to test (e.g., to determine whether particular debugging traces should be produced), but which play no role in the normal functioning of these procedures and are rarely modified.

(C) They need to be shared between procedures which do not call each other and must be kept alive between successive invocations of these procedures.

(D) They represent constants, too complex to be set up conveniently using a CONST declaration (see Section YYY), which need to be used whenever a procedure is invoked.

(E) They need to be accessible to all logical copies of a recursive procedure (See Section 4.4).

The -capitalize- function appearing in Section 1 can be used to illustrate point (d). As written, this forms the map

```
capital_of := {['a','A'], ['b','B']. ['c','C'],...,['z','Z']}
```

each time it is invoked. To do this is of course wasteful of computer time. Using the CONST declaration described in Section 6.2 we would instead declare capital_of to be a constant having this value, but to do so we would have to write out all the elements of const_of explicitly, a nuisance since this would require us to type 104 apostrophes, 51 commas, 52 brackets, etc. It is more convenient to declare

```
VAR capital_of;
```

and then to add the instructions :

```
small_letters := 'abcdefghijklmnopqrstuvwxyz' ;
big_letters   := 'ABCDEFGHIJKLMNOPQRSTUVWXYZ' ;
capital_of    := {[1,big_letters(i)]: 1=small_letters(i)} ;
```

as part of a main program block before the first use of -capitalize-. The -capitalize- function reduces to the following simple form :

```
PROC capitalize(s);
```

```
        RETURN +/[capital_of(let) ? let : let = s(i)] ;

        END PROC capitalize ;
```

4.3 Programming examples

4.3.1 The 'Buckets and Well' problem: a simple 'artificial intelligence' example.

The following kind of problem, often called the 'buckets and well' puzzle, commonly appears on intelligence tests. Suppose that one is given several buckets of various sizes, and that a well full of water is available. To focus on a simple specific case, suppose that just two buckets, a 3 quart bucket and a 5 quart bucket, are given, and that we are required to use them to to measure out exactly three quarts of water. Since exactly this amount of water to be measured out, no non-precise operation is allowed. This means that only three kinds of operations can be used in a solution of this problem: s.nf

(a) any bucket can be filled brim-full from the well;

(b) any bucket can be emptied completely;

(c) any bucket can be poured into any other, until either the first bucket becomes completely empty or the second bucket becomes brim-full.

As an example, the following is a way of measuring out exactly 4 quarts using only a 3 and a 5 quart bucket.

(i) Fill the 5 quart bucket.

(ii) Pour the 5 quart bucket into the 3 quart bucket (leaving 2 quarts in the 5 quart bucket.)

(iii) Empty the 3 quart bucket.

(iv) Pour the contents of the 5 quart bucket into the 3 quart bucket. (Now 2 quarts are in the 3 quart bucket, and the 5 quart bucket is empty).

(v) Fill the 5 quart bucket.

(vi) Pour the 5 quart bucket into the 3 quart bucket, until the 3 quart bucket becomes full. (This leaves exactly 4 quarts in the 5 quart bucket.)

(vii) Empty the 3 quart bucket.

(Now exactly 4 quarts have been measured out).

The fact that it is easy to program a computer to solve problems of this kind might be considered surprising, since such solutions are often considered to require 'intelligence'. Nevertheless a systematic approach is not hard to find. The key idea is that of 'state'. Specifically, as one moves through the steps of any solution to this kind of problem, the objects being manipulated (in this case, the buckets) will at any moment be in some particular condition. In the case we consider, this 'condition' or 'state' is determined by the amount of water in each of the buckets. We can represent this state as a tuple, of as many components as there are buckets. Initially, when both buckets are empty, the state is [0,0]. The 'target' state for the example considered above is that in which exactly four quarts have been measured into the 5 quart bucket; this is represented by the tuple [0,4]. The state in which both buckets are completely full is [3,5], that in which the three quart bucket is full and the 5 quart bucket is empty is [3,0], etc. In this representaton, the problem solution given by (i-vii) above would be represented as the following sequence of states:

```
[0,0],[0,5],[3,2],[0,2],[2,0],[2,5],[3,4],[0,4]
```

This way of looking at the problem makes it plain that what we need to consider is the set of all possible states, and the manner in which new states can be reached from old. Suppose that the tuple -state- represents the amount of water currently in the buckets, so that state(i) is the amount of water in the i-th bucket, and that the tuple -size- represents the sizes of all the given buckets, so that size(i) is the capacity of the i-th bucket. In the buckets-and-well problem, only the three manipulations (a), (b), and (c) are allowed. If bucket i is poured into bucket j until either i becomes empty or j becomes full, then the amount poured will be

```
state(i) MIN (size(j)-state(j)).
```

Hence the following procedure returns the collection of all states than can be reached in a single step from an initially given state:

```
PROC new_states_from(state);

RETURN {empty(state,j): j IN [1..#state]}
      + {fill(state,j): j IN [1..#state]}
      + {pour(state,i,j): i IN [1..#state], j IN [1..#state]|(i/=j)};

END PROC new_states_from(state);

PROC empty(state,j);  $ empties bucket j

state(j):=0;
RETURN state;

END PROC empty;

PROC fill(state,j);  $ fills bucket j

state(j):=size(j);  $ the 'size' tuple is assumed to be global
RETURN state;
```

```
END PROC fill;

PROC pour(state,i,j);  $ pour bucket i into bucket j

amount := state(i) MIN (size(j)-state(j));  $ amount that can be poured
state(i) -:= amount; state(j) +:= amount;

RETURN state;

END PROC pour;
```

We can now solve our problem by a systematic process of state exploration. We start in the initial 'all buckets empty' state ([0,0] in the preceding example). Next we use the new_states_from routine to generate all the states which can be reached in one step from this starting state. Then we generate all states which can be reached in one step from these second level states, etc. States which have been encountered previously are ignored; the ones which remain are precisely those which can be reached from the start in two steps but no fewer. From these, we generate all states which can be generated in three steps but no fewer, and so forth. As we go along, we check to see if the target state has yet been reached. Eventually, we either reach the target state, thereby solving our problem, or find that no new states can be generated, even though the target state has not been reached. In this latter case, the problem clearly has no solution.

The following figure illustrates the notion of state-search and shows some of the states that come up during search for a solution of our two-bucket example:

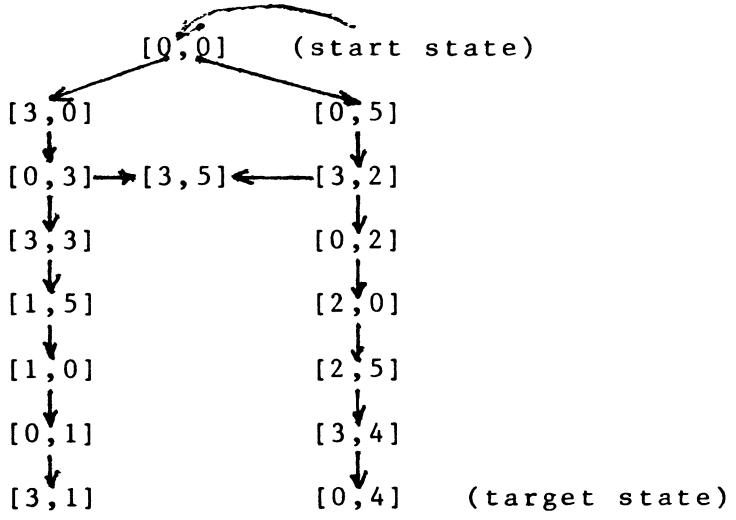


Figure 4.3. States of a two-bucket problem; bucket sizes are [3,5]

Note that in this figure we only show transitions which lead to states that have not been seen before. Other transitions are redundant, since the shortest path from start state to the target state will never pass through the same state twice.

To be sure that we can reconstruct the path from start to target once the target has been reached, we proceed as follows. Whenever a new state ns is seen for the first time it will have been generated from some immediately preceding old state os. As states are generated, we keep a map came_from which maps each new state ns into the old state os from which ns has been reached. Once the target state has been reached, we can use this map to chain back from the target to the start state. Then the desired soultion is simply the reverse of the sequence thereby generated.

The following code implements this state-generation and backchaining procedure. It is deliberately written in a manner which hides all information concerning the structure of states, as well as all details concerning the way in which new states arise from old. This makes it possible to use the same routine to solve many different kinds of state-exploration problems.

```
PROC find_path(start,target);    $ general state-exploration procedure.

came_from := {[start,start]};    $ the start state is considered
                                  $ to have been reached from itself
just_seen := {start};            $ initially, only the start
                                  $ state has been seen
(WHILE just_seen/={})            $ while there exist newly seen states
  brand_new := { };              $ look for states that have not
                                  $ been seen before
  (FOR old_state IN just_seen, new_state IN new_states_from(old_state)
                          |came_from(new_state)=OM)

    brand_new WITH:=new_state;      $  record a brand_new state
    came_from(new_state) := old_state; $ and record its origin

    IF new_state=target THEN GOTO got_it; END;
                    $ since problem has been solved
  END FOR;

  just_seen := brand_new; $ now the brand-new states define those which
                           $ have just been seen

END WHILE;

RETURN OM; $ at this point all states have been explored, and the
           $ target has not been found, so we know that no solution
           $ exists.

got_it:    $ at this point the target has been found, so we chain
           $ back from the target to reconstruct the path from start
           $ to target

   rev_path := [target]; $ initialize the path to be built

(WHILE (last_state :=  rev_path(#rev_path))/=start)
    rev_path WITH:= came_from(last_state);
END WHILE;

RETURN [rev_path(j): j IN [#rev_path,#rev_path-1..1]];
```

```
    END PROC find_path;
```

The following main program can be used to acquire a problem-specification interactively and to invoke the find_path routine to solve it. Again we hide all problem-specific information in appropriate subroutines.

```
    VAR
        size;  $ global variable for storing problem specification

    (WHILE (prob_specs:=get_prob_specs( ))/=OM)

        [start, target, size] := prob_specs;

        IF (path := find_path(start,target))=OM THEN
            print('This problem is definitely unsolvable')
        ELSE
            print('The following sequence of states constitutes a'
                                    solution:');
            (FOR x IN path) print(x); END;
        END IF;

     END WHILE;
```

The parameterless get_prob_specs function interrogates the user to acquire the description of a particular buckets-and-well problem to be solved. (Note that we assume here that the program we have been considering is being run interactively from a terminal.)

```
    PROC  get_prob_specs;  $ acquires and returns specifications
                           $ of problem
    LOOP DO

print('Enter a tuple to define bucket sizes, or type ''STOP'' to halt:');

    read(x);
    IF x='STOP' THEN QUIT; END;

print('Enter a tuple of the same length to define initial bucket states:');

    read(y);
    IF y='STOP' THEN QUIT; END;

print('Enter a tuple of the same length to define target of problem:');

    read(z);
    IF z='STOP' THEN QUIT; END;
    data := [y,z,x];
    IF EXISTS t=data(i) | (NOT IS_TUPLE(t)  OR #t /= #data(1)
         OR EXISTS c=t(i) | NOT IS_INTEGER(c) OR c < 0)   THEN

print('Illegal problem specification.' 'Please re-enter or type ''STOP''

                                        to halt.');
```

```
        CONTINUE;  $loop, to try again
    ELSE

        RETURN data;
    END IF;
END LOOP;
RETURN OM;    $ Since this point will only be reached if 'STOP'
              $ has been typed.
END PROC get_prob_specs;
```

Since the notion of 'problem state' used in the foregoing is quite general and since we have written the find_path procedure and the main program block shown above in a manner which insulates them from the details of the problems that they solve, we can use these procedures to handle any path-finding problem of the same general class as the 'buckets and well problem'. Another amusing problem of this kind is the 'goat, wolf, and cabbage' puzzle. In this puzzle, a man, who brings with him a goat, a wolf, and a cabbage, comes to a river which he must cross in a boat just large enough for himself and one but not two of the objects goat, wolf, and cabbage. He can never leave the goat and wolf, or the cabbage and goat, alone together, since in the first case the wolf would eat the goat and in the second the goat would eat the cabbage. How is he to cross the river?

To develop a program which solves this puzzle, we have only to rewrite the -new_states_from- function and the parameterless-get_prob_specs-procedure. First, we need to decide on a representation of the states of the puzzle. We can designate the four objects appearing in the puzzle by their initials as 'G', 'W', 'C', and 'M' (man) respectively, and represent each state of the puzzle by a pair [l,r], where -l- designates the set of all objects remaining to the left of the river, and -r- designates the set of all objects that have been moved across the river. For example,

```
[{'G', 'M'}, {'W'.'C'}]
```

represents the state in which the wolf and the cabbage have been moved across, and the man has returned to the left side of the river to get the goat. The start state is then

```
[{'G', 'W', 'C', 'M'}, { }]
```

and the target state is

```
[ { }, {'G', 'W', 'C', 'M'}]
```

The -new_states_from- procedure appropriate for this problem can be represented as follows:

```
PROC new_states_from(state);

[l,r] := state;  $ 'unpack' state into its 'left' and 'right' portions

RETURN IF 'M' IN l THEN {[l-{'M',x}, r + {'M',x}]:
                         x IN l| x/='M' AND is_legal(l-{'M',x})}
```

```
    ELSE
  {[1 + {'M',x}, r-{'M',x}]: x IN r| x/='M' AND is_legal(r-{'M',x})}
END;

END PROC new_states_from;

PROC is_legal(s);  $ test to see if s meets conditions of puzzle

RETURN NOT ({'G', 'C'} SUBSET s OR {'G', 'W'} SUBSET s);

END PROC is_legal;
```

4.4 Recursive Functions.

The value that a mathematical function f(x) of an integer, tuple, or set variable takes on for a particular x can often be expressed in terms of the value of the same function for smaller argument values x'. Several examples of this general principle are:

(i) The 'factorial' function n!, equivalent to */[i:i in [1..n]], satisfies the identity

```
n! = IF n = 1 THEN 1 ELSE n*((n-1)!) END
```

(ii) The sum sigma(t)=+/t of all the components of a tuple t satisfies the identity

```
sigma(t) = IF  t=0 THEN OM ELSEIF  t=1 THEN t(1)
                   ELSE t(1) + sigma (t(2..))  END;
```

(iii) The tuple sort(s) representing the elements of s in sorted order satisfies the identity

```
sort(s) = IF  s=0 THEN [ ] ELSE [MIN/s] + sort(s-{MIN/s}) END
```

This same function sort(s) also satisfies many other interesting identities. Suppose, for example, that we pick an arbitrary element x from the set s, and then divide the remaining elements of s into two parts, the first, L, containing all elements less than x, the second, G, containing all elements greater then x. Then if we sort the elements of L and G, and concentrate the resulting sorted tuples, sandwiching x between them, we clearly get a tuple t which contains al the elements of s in sorted order. This shows that the function sort(s) also satisfies the identity

```
sort(s) = IF (x:= ARB s)= OM THEN [ ] ELSE
      sort({y IN s:y<x}) +[x] + sort({y IN s:y>x}) END;
```

Identities of the kind appearing in the preceding examples are called recursive definitions and the functions appearing in them are called recursively defined functions. Such recursive definitions all have the following features:

(a) For certain particular simple or 'minimal' values (like n=1 in (i) or t=[ ] in (ii)) of the argument variable x of a recursively defined function f(x), the value of f(x) is defined explicitly.

(b) For all other argment values x, the value of f(x) is expressed in terms of the value that f takes on for one or more smaller argument values x1,x2,..xn. That is, there exists a relationship of the general form

f(x) = some_combination(f(x1),f(x2),..,f(xn))

(c) Repeated use of the relationship (b) will eventually express any value f(x) in terms of various values f(y) each of which has a parameter y which is minimal in the sense of (a), so that all values f(y) in terms of which f(x) is ultimately expressed are known explicitly.

Any recursive relationship satisfying (a,b,c) gives a method for calculating f(x) for each allowed argument x. Like many other programming languages, SETL allows one to express such recursive calculations very simply and directly, by writing recursive functions, i.e. functions which invoke themselves. This can be done for each of the three examples given above, which then take on the following forms:

```
PROC factorial(n);   $ calculates the factorial n!

RETURN IF n=1 THEN 1 ELSE n*factorial(n-1) END;

END PROC factorial;

PROC sigma(t);     $ calculates +/t

RETURN IF  t=0 THEN OM ELSEIF  t=1 THEN t(1)
               ELSE t(1) + SIGMA (t(2..)) END;

END PROC sigma;

PROC sort(s);  $ recursive sorting procedure

RETURN IF s ={ } THEN [ ]
       ELSE [MIN/S] + sort1(s LESS MIN/S) END;

END PROC sort;
```

These examples illustrate the following general remarks concerning recursive procedures:

(i) Syntactically, recursive functions (and procedures) have the same form as other functions and procedures. The only difference is that recursive procedures invoke themselves, while other procedures happen not to.

(ii) The same name-scoping rules apply to recursive as to other procedures.

Note that a recursive function f(s) uses itself, but always applies itself to arguments smaller than s; this is why the calculation of f eventually terminates.

Recursive functions f need not invoke themselves directly: They can invoke other functions g which invoke f, or g can invoke some h which then invokes f, etc. A group of functions which invokes each other is sometimes called a mutually recursive family of functions, and any function belonging to such a mutually recursive family is itself called recursive.

For an example of such a mutually recursive family, consider the problem of defining an overal order for SETL objects, which will allow any two SETL objects to be compared to each other. (Such an order could, for example, serve as the basis for an output routine which always arranged the elements of sets in increasing order, thereby making it easier to locate elements in large sets when they were printed.) To define such an order, we can agree on the following conventions:

(a) OM always comes first, integers before reals, reals before strings, strings before atoms, atoms before tuples, and tuples before sets. (Atom is a SETL data-type. For more on atoms see section 5.3).

(b) Among themselves, integers and reals are arranged in their standard order, strings in their standard alphabetical order, and atoms in the order of their external printed representations, i.e. if x and y are two atoms then x comes before y if and only if (STR x) < (STR y). (Note that the STR x operator produces a string identical with the external printed form of the object x; see Section 2.1.)

(c) Tuples are arranged in lexicographic order, i.e. t1 comes before t2 if, in the first component in which t1 and t2 differ, t1 has a smaller component than t2.

(d) To compare two sets, first arrange their elements in order. This allows them to be regarded as tuples; then apply rule (c).

The following mutually recursive group of functions implements the ordering strategy we have just described.

```
PROC is_bigger(x,y);  $ return TRUE if x>y in the
                      $ order just described

RETURN IF x=y OR y=OM THEN TRUE
ELSEIF x=OM THEN FALSE
ELSEIF TYPE x /= TYPE y THEN type_number(x) > type_number(y)
ELSEIF TYPE x = 'ATOM' THEN STR x > STR y
ELSEIF TYPE x= 'TUPLE' THEN lex_compare(x,y)
ELSE lex_compare(sort(x),sort(y));      $ x and y are sets

END PROC is_bigger;

PROC sort(s); $sorts the elements of the sets into the order defined
              $ by is_bigger
              $ A sorted tuple is returned.  The 'bubble' method
              $ is used for sorting
```

```
t:=[x IN s];  $ arrange the elements of s as
              $ a tuple in arbitrary order

(WHILE EXISTS i IN [1.. t-1] | is_bigger(t(i),t(i+1)))
     [t(i),t(i+1)]:=[t(i+1),t(i)];
END WHILE;

RETURN t;

END PROC sort;

PROC lex_compare(t1,t2); $ compare two different tuples,
                         $ in their lecicographic order,
                         $ components being compared by is_bigger

RETURN EXISTS c1=t1(i) | is_bigger(c1,t2(i));

END PROC lexcompare;

PROC type_number(typ);  $ converts typ, which is the name of
                        $ a valid SETL type , into an integer

tno:= {['INTEGER',1],['REAL',2],['STRING',3],['ATOM',4],
       ['TUPLE',5],['SET',6]};

RETURN tno(typ);

END PROC type_number;
```

Until now we have regarded recursive SETL functions simply as SETL representations of recursive mathematical relationships, and have ignored the question of how they are implemented, i.e. how the calculations which they define are actually performed. This is actually the best way to look at the matter, since the calculation used to evaluate a recursive function can be complex and tricky to follow even when the mathematical relationship on which it is based is simple and easy to understand. Nevertheless one occasionally needs to understand how recursive calculations are performed. For example, when an incorrectly programmed recursive procedure malfunctions, one needs to know what is happening in order to diagnose the problem and correct it.

Implementation of recursive functions, like that of mutually recursive groups of functions, is based upon the following rule. Whenever a function f invokes itself, a new logical copy of the function is created, initial parameter values are passed to this new logical copy, and execution of this new logical copy begins with its first statement. While the new copy of f is executing, the old copy of the function f, from which the new copy was created, remains in existence, but execution of it is suspended. The new copy can in turn invoke f, thereby creating a third copy of f, which can even go on in the same way to create yet a fourth copy, etc. However, if the recursion has been written correctly, the arguments x passed to thee successive copies of f will be getting smaller and smaller. Eventually one of them will get small enough for the corresponding value f(x) to be evaluated directly. Once this happens, the currently active copy of the function f will execute a statement

RETURN e

for some directly evaluable expression e. This will pass the value of e back to the place from which the current copy of f (call it CCF) was invoked. CCF will then become superfluous and will disappear. The immediately prior copy of f will then become active, and when it finishes its execution it will in turn pass a value back to the copy of f from which it has been invoked and disappears, etc. Eventually a value, and control, will be returned to the very first copy of f, and the whole recursive evaluation will be completed as soon as this first copy executes a RETURN statement.

As an example of this process of recursive evaluation, suppose that the recursive -sort- routine shown earlier in this section is invoked, and that initially the argument value {30,0,60,40} is transmitted to it. This will trigger the following steps of recursive evaluation.

(i) Copy 1 of -sort- begins to evaluate sort({30,0,60,40})

(ii) The minimum element 0 is removed from the set s, and -sort- is invoked recursively to evaluate sort({30,60,40})

(iii) Copy 2 of -sort- begins to evaluate sort({30,60,40})

(iv) The minimum element 30 is removed from the set s, and -sort- is invoked recursively to evaluate sort({60,40})

(v) Copy 3 of -sort- begins to evaluate sort({60,40})

(vi) The minimum element 40 is removed from th set s, and -sort- is invoked recursively to evaluate sort({60})

(vii) Copy 7 of -sort- begins to evaluate sort({60})

(viii) The minimum (and only) element 60 is remoed from the set s, and -sort- is invoked recursively to evaluate sort({ }).

(ix) copy 5 of -sort- immediately returns [ ] as the value of sort({ }) to copy 4, and disappears.

(x) Copy 4 of sort appends the returned value [ ] to [60], returns the result [60] to copy 3, and disappears.

(xi) Copy 3 appends the returned value [60] to [40], returns the result [40,60] to copy 2, and disappears.

(xii) Copy 2 appends the returned value [40,60] to [30], returns the result [30,40,60] to copy 1, and disappears

(xiii) Copy 1 appends the returned value [30,40,60] to [0], and returns [0,30,40,60], as the final result of the whole recursive evaluation, to the place from which -sort- was first invked.

The complexity of this sequence of steps underscores the fact that whenever possible a recursive SETL function like -sort- should be

looked at as the transcription of a recursive mathematical relationship in this case, the very obvious relationship

```
sort(s) = IF s={ } THEN [ ] ELSE [MIN/s] + sort(s LESS MIN/s) END;
```

rather than in terms of the sequence of steps required for its evaluation. However, the way in whch recursive routines are evaluated becomes relevant if they are miswritten and consequently malfunction. Certain common pathologies are associated with malfunctioning recursive routines and one needs to be able to recognize them when they appear. A common error is to write a recursion which does not handle its easy, directly evaluable cases correctly, or which for some reason never reaches a directly evaluable case. If this happens, a recursive routine will create more and more copies of itself without limit, until the entire memory of the computer on which it is running is exhausted, and a final, 'MEMORY OVERFLOW' error message is emitted.

In somewhat more complex cases, a malfunctioning recursive function will loop indefinitely, first creating additional copies of itself, then returning from and erasing these, then again creating new copies of itself, again returning from and erasing these, etc., without any overall progress to termination. Such a 'nonterminating recursive loop' is likely to produce much the same symptoms as a nonterminating WHILE loop, namely the program will run on, either with no output or with a flood of repetitive output, until the operating system notices that it has outrun its time limit and terminates it forcibly. This situation is most easily diagnosed at an interactive terminal, simply by printing out the parameters transmitted to the recursive function each time it is invoked; this pattern of parameters will fail to show the logical pattern upon which your hopes for eventual termination of the recursion rest.

Having said all this, we now go on to describe another interesting recursive procedure, namely

4.4.1 Robert Floyd's Quicksort Procedure.

Quicksort: This sorting method, due to Robert Floyd of Stanford University, works as folows: If the tuple t of elements to be sorted has no elements or just one element, we have nothing to do, since an empty tuple or a tuple with just one element is always sorted. Otherwise, we remove the first element x from t, and divide what remains into two parts, the first ('small_pile') consisting of all those component smaller then x, the second ('large_pile') consisting of all those components at least as large as x. We then sort these two piles separately. This can most readily be done just by using quicksort itself recursively. Finally, we recombine to get all the original components in their sorted order. This is done by putting the sorted smallpile first, followed by the element x, and then followed by the sorted largepile.

See the attached figure for further explanation of the way in which quicksort works. Code for this procedure can be written as follows:

```
PROC quick_sort(t);  $ Floyd quicksort procedure, first form

IF #t<2 THEN RETURN t; END;
```

```
x := t(1) ;                              $ take the first component
small_pile := [y : y=t(i) | y < x] ;
large_pile := [y : y=t(i) | y >=x and i > 1] ;

RETURN quick_sort(small_pile) + [x] + quick_sort(large_pile) ;

END PROC quick_sort ;
```

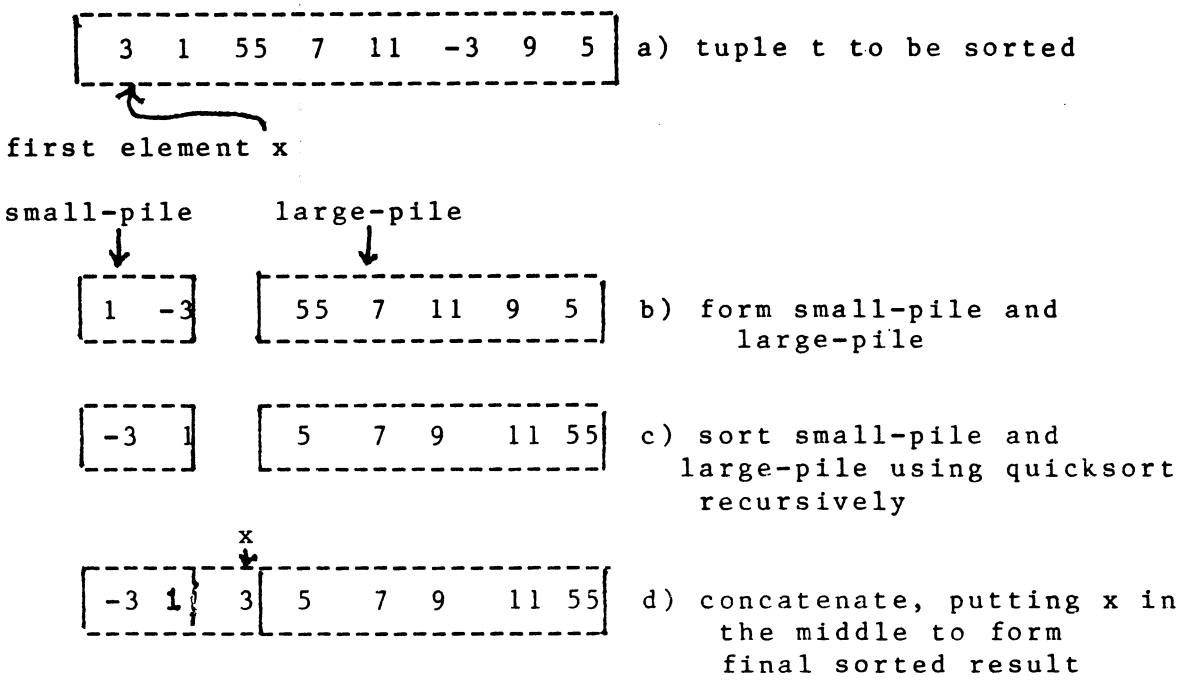


Figure 4.4: Robert Floyd's Quicksort Procedure

By using SETL expression features more strenuously, we can write this whole procedure in just one statement, namely as

```
PROC quick_sort(t);   $ Floyd quicksort procedure, second form

RETURN IF #t < 2 THEN t
    ELSE
      quick_sort([y : y=t(i) | y < t(1)]) + [t(1)]
    + quick_sort([y : y=t(i) | y >= t(1) and i>1] )
    END ;

END PROC quick_sort;
```

4.4.2. Another Recursive Procedure: Mergesort

The 'quicksort' procedure that has just been presented sorts by separating the array to be sorted into two piles which can be sorted separately and then combined. This recursive approach, sometimes called 'divide and conquer', forms the basis for many efficient data-manipulation algorithms. It is often most effective to divide the problem given originally into exactly two halves of equal size. 'Quicksort' does not always lead to this equal division, since random selection of a component x of a tuple t may cause it to be divided into parts [y:y IN t | y < x] and [y:y IN t | y > x] which are very different in size. For this reason, we will now describe another recursive sorting technique, called mergesort, which does begin by dividing the tuple t that is to be sorted into two parts of equal size. This procedure works as follows:

(i) Divide t (at its middle) into two equal parts t1 and t2, and sort them separately.

(ii) Then merge the two sorted parts t1, t2 of t, by removing either the first component of t1 or the first component of t2, whichever is smaller, and putting it first in the sorted version of the full tuple t, after which we can continue recursively, merging the remaining components of t1 and t2.

Code for this procedure is as follows:

```
PROC sort(t);  $ recursive mergesort procedure

RETURN IF #t < 2 THEN t
    $ since a tuple of length 0 or 1 is ipso facto sorted
    ELSE merge(sort(t(1..#t/2)), sort(t(#t/2+1..)))  END;

END PROC sort;

PROC merge(t1,t2);  $ auxiliary recursive procedure for merging

RETURN IF t1=[ ] THEN t2
    ELSEIF t2=[ ] THEN t1
    ELSEIF t1(1) < t2(1) THEN [t1(1)] + merge(t1(2..),t2)
    ELSE t2(1) + merge(t1,t2(2..)) END;

END PROC merge;
```

Instead of programming the -merge- procedure recursively, we can write it iteratively. For this, we have only to work sequentially through the two tuples t1 and t2 to be merged, maintaining pointers i1, i2 to the first component of each which has not yet been moved to the final sorted tuple t being built up. Then we repeatedly compare t1(i1) to t2(i2), move the smaller of the two to t, and increment the index of the component that has just been moved to t. This revised merge procedure is as follows:

```
PROC merge(t1,t2);  $ iterative variant of -merge- procedure

t := [ ];              $ merged tuple to be built up
i1 := i2 := 1;         $ indices of first components not yet moved

(WHILE i1 < #t1 AND i 2 < #t2)
   IF t1(i1) < t2(i2) THEN $ 'move ' t1(i1) to t
       t WITH:= t1(i1);
       i1 +:= 1;
   ELSE                    $ 'move' t2(i2) to t
       t WITH := t2(i2);
       t2 +:= 1;
   END IF;

END WHILE;

RETURN t + t1(i1..) + t2(i2..);
$ note that at most one of t1(i1..) and t2((i2..) is non-null
```

```
END PROC merge;
```

4.4.3. Binary searching: a fast recursive searching technique.

If the components of a tuple t are arranged in random order, then to find the component or components having a given value we must search serially through every one of the components of t; clearly no component of t can go unexamined, since this may be precisely the component we are looking for. On the other hand, if the components of t are numbers or character strings, and if they are arranged in sorted order, then, as every one who has ever looked up a word in a dictionary or a name in a telephone book should realise, a much faster searching procedure is available. The most elegant expression of this searching procedure is recursive, and is as follows:

(i) Compare the item x being sought to the middle item t(#t/2) of the sorted tuple t. If x is greater than (resp. not greater than) this middle item, proceed recursively to search for x in the upper (resp. lower) half of t.

(ii) The search ends when the vector in which we are searching has length equal to 1.

In coding this procedure, we maintain two quantities -lo-,-hi-, which are respectively the low and the high limits of the zone of t in which we must still search. When the search procedure is first called, lo should be 1 and hi should be #t. When -lo- and -hi- become equal, we return their common value. If this locates a component of t equal to x, we have found what we want; otherwise we can be sure that x is not present in t, i.e. that no component of t is precisely equal to x.

Recursive code for this searching procedure is as follows:

```
PROC search(x,t,lo,hi);
    $ binary search for x in t between -lo- and -hi-

RETURN IF lo=hi THEN lo
    ELSEIF x <= t(mid := (lo+hi)/2) THEN search (x,t,lo,mid,hi)
    ELSE search (x,t,mid+1,hi) END;

END PROC search;
```

It is easy to express this search iteratively rather than recursively, namely we can write

```
PROC search((x,t);  $ iterative form of binary search procedure

lo := 1; hi := #t;  $ initialise search limits

(WHILE lo < hi)

    IF x <= t(mid := (lo+hi)/2) THEN
        hi := mid;
    ELSE
        lo := mid+1;
```

```
        END IF;
    END WHILE;
    RETURN lo;
    END PROC search;
```

Binary searching can be enormously more efficient than simple serial searching. Suppose, for example, that the sorted tuple t to be searched is of length one million. Then to search t serially several million elementary operations will be required. On the other hand, since 1,000,000 is roughly 2**20, only twenty probes will be required to locate a component of t by binary searching. Hence, for sorted tuples of this length, binary searching is roughly 50,000 times as fast as serial searching. This illustrates the vast efficiency advantage that can be gained by proper choice of the algorithm that you will use.

4.4.4 The 'Towers of Hanoi' Problem

Among the many different kinds of puzzles that can be bought in toyshops, the 'Towers of Hanoi' puzzle is a classic. This puzzle involves a board with three identical pegs and a set of rings of decreasing external diameter which will fit snugly around any of the pegs. As initially set up, the puzzle looks like this:

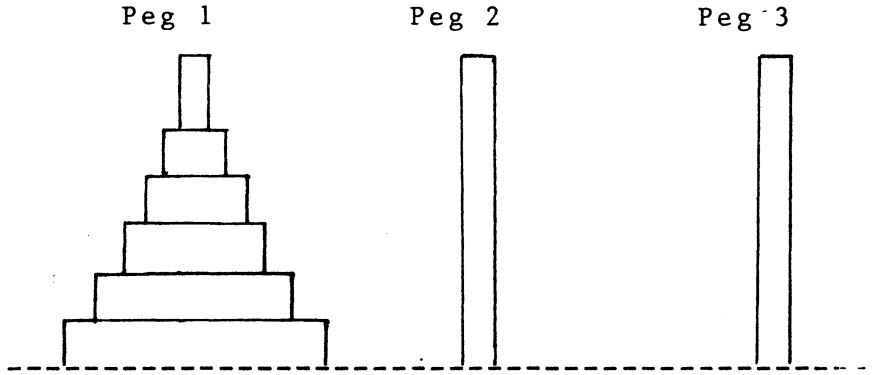


Figure 4.5: The 'Towers of Hanoi' Problem

To solve the puzzle one must move all the disks from the particular peg (peg 1) on which they are originally placed to one of the other pegs (say, to peg 3). However, only one disc can be moved at a time, and it is forbidden to ever place a larger disc on top of a smaller disc.

Recursion gives us an amazingly effective way of writing a solution to this problem. The key idea is this: since a large disk can never be placed atop a smaller, all the disks except the bottom one must be moved to peg 2 before we can move the bottom disk from peg 1 to peg 3. Hence, to move a pile of n disks from peg 1 to peg 3, we must

(a) move a pile of (n-1) disks from peg 1 to peg 2;

(b) move the n-th disk from peg 1 to peg 3
(c) move a pile of (n-1) disks from peg 2 to peg 3

The following elegant recursive function generates the sequence of moves required; each move is represented as a pair [f,t] showing the pegs from which and to which a peg is moved.

```
PROC moves(ndisks, fr, to, via); $ moves n disks from peg -fr- to
                                 $ peg -to-

RETURN ( IF ndisks=1 THEN [[fr,to]]
       ELSE moves(ndisks-1, fr, via, to) + [[fr,to]]
            + moves(ndisks-1, via, to, fr) END ) ;
END PROC moves;
```

A function is always sent some collection of parameter values, and calculates a single result value, which it RETURNs, from them. Occasionally, however, one wants to use PROCEDUREs in a somewhat different way, namely, one wants to invoke a procedure expressly in order to modify some object that already exists. In this case, such a procedure is invoked for its effect, rather than for the value it delivers. This use of procedures moves us away from the notions of 'value' and 'expression' and focuses more on the somewhat different notion of program state, i.e., the collection of all values local and global variables have at each moment during a computation. What we will be describing now is the way in which procedures are used to modify this program state. There are two ways in which procedures can have this effect: one of them is to construct procedures which modify one or more of their calling parameters; the second is to have a procedure modify one or more global variables. We shall examine each of these possibilities separately.

4.5 Procedures which modify their parameters

A function is always sent some collection of parameter vales, and always calclates some single value, which it returns, from them. Occasionally, however, one wants to use PROCEDURES in a somewhat different way. More specifically, one wants to transmit (zero or more) arguments to them, but thhen to have the procedure modify some or all of tis parameters, after which it must make these modified values available to the code which has invoked it. A related possibility is to invoke a procedre simply in order to modify one or more globally available variables. (See Section 4.5).

This use of procedures is perfectly legal in SETL, and is accomplished as follows. A procedure's header line lists its parameters, as for example in

```
PROCEDURE my_proc(x,y,z);
```

Parameters listed in this way can be modified within the body of the procedure (i.e., within -my_proc-) but parameter values are ordinarily local to the
procedure, so that these modifications are not be transmitted back to the point from which the procedure was invoked. For example,

if we define the procedure

```
PROC change_parameter(x);

x:=0;

RETURN x;

END PROC change_parameter;
```

and invoke it by

```
(2)     y := 1 ;
        z := change_parameter(y);

        print('z is:', z, 'y' is:,y);
```

then the -print- statement will produce the output

```
z is: 0  y is: 1
```

This reflects the fact that the RETURN statement in the PROC returns the final value of the variable x (which is local to the PROC), but that modifications to the procedure parameter x are not transmitted back to the point of invocation and therefore do not affect the value of the actual argument y appearing in (2). Thus the argument y remains unchanged.

This is the rule which ordinarily applies to PROCs, and which is most appropriate for PROCs used as functions. However, it is possible to bypass this rule, and to create PROCs which do modify one or more of the actual arguments with which they are invoked. To do this, one simply prefixes the 'parameter qualifer' RW (meaning 'read/write parameter') to each parameter corresponding to one of these modifiable arguments. Suppose, for example, that we modify the procedure (1a), making it

```
(1b)    PROC change_parameter(RW x);

        x:=0;

        RETURN x;

        END PROC change_parameter;
```

Then the output of the -print- statement in (2) will change to

```
z is: 0 y is: 0,
```

reflecting the fact that now changes in the value of the parameter x of the PROC (1b) will be transmitted back to the point from which the PROC was invoked.

PROCs whose parameters are qualified in this way will generally not be used as functions that return values (though technically it is legal to use them as functions). Instead, they will ordinarily be invoked simply by writing their names followed by their actual argument lists, as is

illustrated by

```
(3)     y:=1;
        change_parameter(y);
        print('y is:',y);
```

which produces the output

```
        y is: 0
```

Any procedure my_proc(xl,..,xn) can be invoked in this way, i.e. simply by writing a statement of the form

```
(4)     my_proc(al,...,an);
```

where al,...,an is any list of expressions (called, as usual, the 'actual arguments' of the invocation (4)). An invocation like (4) is logically equivalent to an invocation

```
(4b)    junk_variable := my_proc(al,...,an);
```

where -junk_variable- can be the name of any variable whose value is never used for anything else.

Of course, if the procedure -my_proc- invoked by (4) does not modify any of its arguments, an invocation like (4) will generally not be very useful, since none of the arguments al,...,an will change and since the value returned by -my_proc- is simply thrown away. On the other hand, if the procedure -my_proc- does modify its arguments, then the invocation (4) will trigger corresponding modifications of any arguments ag which correspond to parameters carrying the qualification RW.

PROCs which modify some of their arguments and which are normally invoked in this way are often called 'simple- procedures', as distinct from 'functions', i.e. from PROCs which do not modify their arguments and are normally invoked in the manner illustrated by

```
        x := my_function(al,...,an);
```

Since the value RETURNed by a simple-procedure will just be thrown away, the expression e appearing in a statement

```
        RETURN e;
```

within such a procedure is usually without significance and may as well be OM. SETL allows

```
        RETURN OM;
```

to be abbreviated simply as

```
        RETURN;
```

and this is the form of the RETURN statement which is appropriate to use in simple-procedures. Note also that a RETURN statement immediately preceding the trailer line of a simple-procedure can be omitted.

Simple-procedures with no parameters can be invoked just by writing their names followed by a semicolon, as in

```
my_simple_proc_without_parameters ;  $ invokes procedure with
                                     $ this name.
```

As an example, here is a simple-procedure which 'compresses' a tuple by dropping out all of its OM compnents:

```
(5a)    PROC compress (RW t);

        t := [x IN t | x/=OM];

        END PROC compress;
```

(Here we have made use of one of the rules stated above to save writing a RETURN statement just before the trailer line of this PROC.)

Note that if x initially has the value [1,OM,OM,OM,2,OM,3], then the invocation

```
(6b)    compress(x);
```

will give x the value [1,2,3].

As a matter of style, note also that instead of writing (5a) we could have written a closely related function, namely

```
(5b)    PROC compress (t);

        RETURN [x in t | x/=OM];

        END PROC compress;
```

in which case would have had to write

```
(6b)    x := compress(x);
```

to get the effect of (6a). The form (6a) is sometimes slightly more convenient to write and it is this convenience that can induce us to write a simple-procedure rather than a function for some purpose we have in mind.

In addition to the parameter qualifier RW, two additional qualifiers RD and WR are provided. In general, parameter qualifiers have the following significance:

RD read parameter: can be read and written within its PROCEDURE, but modifications to it will not be transmitted back to the corresponding actual argument.

RW read/write parameter: can be read and written within its PROCEDURE, and modifications to it will be transmitted back to the corresponding actual argument.

WR write-only parameter: can be written and will be transmitted back to the corresponding actual argument, but will not be read.

If none of these qualifiers is attached to a particular procedure parameter, the parameter will be treated as if it were qualified with 'RD'. Thus RD is the 'default' qualifier for otherwise unqualified parameters of procedures.

Next suppose that a procedure called -my_proc- has one (or more) parameters x which are qualified with RW or WR. In this case an invocation

(7a) my_proc(e) ;

of -my_proc- is translated by introducing an otherwise unused 'compiler temporary' variable (call it -var-), and treating (7a) exactly a if it were

(7b)
```
var := e ;
my_proc(var);
e := var;
```

Thus the only forms of expressions which can appear as actual arguments in place of parameters qualified by RW or WR are those which can legally appear to the left of an assignment operator. (See Section 2.12 for a comprehensive discussion of these 'assignment targets'). This means that the invocations

```
my_proc(3) ;
```
and
```
my_proc(x + y);
```

are both illegal, since the assignments

```
3 := var;
```
and
```
x + y := var ;
```

would both be illegal. On the other hand, the invocations

```
my_proc(t(i)) ;  $ where t is a map or tuple
```
and
```
my_proc([x,y]);
```

are both legal, and have exactly the same meanings as

```
var := t(i) ;
my_proc(var) ;
t(i) := var;
```
and
```
var := [x, y] ;
my_proc(var) ;
```

```
[x, y] := var ;
```

respectively.

One final, rather esoteric, point deserves mention. Actual argument values are transmitted to a procedure and become the values of its formal parameters immediately upon invocation of the procedure. These values are transmitted by copying, i.e., each parameter receives a logically independent copy of the appropriate actual argument value upon procedure invocation. If the procedure modifies its parameters, it is these copied values that are modified; the original argument values remain unchanged. Moreover, even if the procedure transmits changes in its parameter values back to the point of invocation, these changes are only transmitted when the procedure executes a RETURN, at which time an assignment like that appearing in (7b) takes place. These rules are natural enough, and normally require little thought. However, examples which show their effects can be contrived. For example, consider the following code, in which the variable y is global :

```
PROGRAM esoteric;

    VAR x,y;        $This declaration makes x and y global

    x := 'initial_val of x,' ; y := 'initial_val_of_y' ;

    manipulate(x,x,y) ;
    print('y is:',y);

    PROC manipulate(u,v,RW w)

    print('u is',u,'v is',v) ;
   $ this will print: u is initial_val_of_x v is initial_val_of_x

    u: = 'changed,';

    print('u is',u,'v is',v);
    $ this will print: u is changed v is initial_val_of_x

    $ Note that u and v remain different even though the
    $ corresponding actual arguments are the same

    w := 'mangled,';

    print('w is',w,'y is',y); $ note that y is global
    $ this will print: w is mangled, y is initial_value_of_y

    $ note that y is still unchanged, even though the change in
    $ w will be transmitted back to y when we return from this PROC

    END PROC manipulate;

END PROGRAM esoteric;
```

Note finally that the last line of output produced by this program, which will be produced by the -print- statement (in line 5 of the program) which immediately follows the invocation of -manipulate- will be

y is mangled

4.6 Exercises

Ex. 1 Write a procedure whose inputs are a tuple t of integers and a tuple s of integers in increasing order, and which returns a tuple t1 of length s+1 defined as follows: the first component of t1 is the number of components of t which are not greater than s(1); for j between 2 and #s, the j-th component of t1 is the number of components of t which are greater than s(j-1) but not greater than s(j); ad the last component of t1 is the number of components of t which are greater than the last component of s. Try to make your procedure efficient.

Ex. 2 'Bags', used in some programming languages, are like sets, but their elements can occur multiply. In SETL, a bag b can be represented in two obvious ways, namely

(a) by a tuple: i.e. the elements of b can be arranged in some arbitrary order, and made the component of a tuple; or

(b) by a map, which sends each element of b into the number of times that it occurs within G.

Write a pair of procedures which convert between these two different representations of a bag G. Also, write a collection of procedures which extend the following set operations to bags in the most useful way:

(i) b1+b2, b1*b2, and b1-b2 (where b1 and b2 are bags)

(ii) x IN b (were b is a bag and x is arbitrary)

Ex. 3 The following table describes the tax due or D dollars of taxable income. Write a procedure which, given D, will return the amount of tax due.

TO BE CONTINUED

Ex. 4 Write a program which will read in a sequence of lines, each containing someone's name, first name first, and print out an alphabetized list of these names, in alphabetic order of last names. Repeat this exercise, but this time print the alphabetized list with last names first.

Exercises On Permutations

A permutation is a one-to-one mapping of a set s of n items into itself. If the set s consists of the integers from 1 to n, then such a permutation can be represented as a vector v of length n such that every integer from 1 to n appears as a component of v. The following exercises concern various properties of permutations.

Ex. 5 The product prod(v1,v2) of two permutations v1 and v2 is the vector v such that v(i)=v1(v2(i)) for each i in {1..#v}. The identity permutation e of n integers is the permutation represented by the vector [1,2,..,n]. The inverse inv(v) of a permutation is the permutation in such that prod(v,inv(v))=e. Write two SETL functions -prod- and -inv- which realize these operations. Write a procedure rand_perm(n) which generates a different random permutation of the integers from 1 to n each time it is called.

Ex. 6 Check the following facts concerning permutations by generating a few random permutations and verifying that each fact asserted holds for these permutations. (The routines described in Exercise 5 should be used for this purpose.)

(a) The product of two permutations is a permutation, and the product of permutations is associative.

(b) prod(inv(v),v)=e for each permutation v.

(c) prod(inv(u),inv(v))=inv(prod(v,u)) for any two permutations u,v of n elements.

(d) Define power(u,k) to be the product of k copies of the the permutation v. Check that power(v,j+k)=prod(power(v,j), power(v,k)). Check that for each permutation v there exists a positive integer k such that power (v,k)=e.

(e) Is prod(u,v)=prod(v,u) true for every pair u,v of permutations of n items?

Ex. 7 A program to generate all permutations (rearrangements) of the integers 1 thru n can be built up as follows. Start with the numbers in the sequence s=[1..n]. Then repeatedly find the last element s(j) in the sequence s such that s(j+1)>s(j). Let s(i) be the last element following s(j) such that s(i)>s(j). Interchange s(i) with s(j), and then reverse the sequence of elements following the j-th position. This gives the next permutations s.

Write this permutation-generation procedure in SETL, and use it to write out the list of all permutations of the integers 1 thru 5. Use this same procedure to create a program which reads in a string of length 5 and prints it out in all possible permutations, but without any repetitions.

Ex. 8 If a second order polynomial P(x)=A*(x**2)+B*x+C with integer coefficients A,B,C has a first-order polynomial M*x+N as a factor, then M is a factor of A and N is a factor of C. Write a procedure which uses this fact to test polynomials like P(x) to see if they can be factored, and which produces the two factors of P if P can be factored. How efficient can you make this factorization procedure?

Can you devise a similar procedure for factoring third order polynomials with integer coefficients?

Ex. 9 As of the present date (early 1981), tokens on the New York City subway system cost 60 cents. Tokens are sold at change booths. Purchasers

normally pay for tokens without saying anything, simply by passing a sum of money to the change booth attendant. Certain sums of money (e.g. $1, which will purchase only one token) are unambiguous. Others, like a five dollar bill, are ambiguous, since they will purchase anywhere from one to eight tokens. On the other hand, $5.50 is unambiguous, since the likely reason for adding the last fifty cents is to pay for nine rather than just eight tokens. Write a program which will read a tuple designating a collection of bills and coins, decide whether this is ambiguous or unambiguous, and print out an appropriate response (which might be either 'How many tokens do you want?' or 'Here are n tokens'.)

Ex. 10 Write a function whose argument is a tuple t with integer or real coefficients and which returns the positions of all the local maxima in t, i.e all the components of t which are larger than either of their neighboring components.

Ex. 11 Before Britain began to use decimal coinage, its money consisted of pence, shillings worth 20 pence each, and pounds worth 12 shillings each. Write a procedure to add sums of money represented in this way, reducing the sum to pounds, shillings, and pence. (Summ of money can conveniently be represented as triples.) Write a procedure that will subtract sums of money represented as pounds, shillings, and pence, and which could have been used to make change in pre-decimal British shops.

Exercises On Recursion

Ex. 12 The greatest common divisor GCD(x,y) of two positive integers is the largest positive integer z such that (x MOD z)=0 and (y MOD z)=0. (If x and y are equal, then GCD(x,y)=x). Write procedures each of which calculates GCD(x,y) efficiently by exploiting one of the following mathematical relationships:

(a) GCD(x,y)=GCD(x-y,y) if x>y
(b) GCD(x,0)=x and GCD(x,y)=GCD(x MOD y,y) if x>y.
(c) GCD(x,y)=2*GCD(x DIV 2,y DIV 2) if x and y are both even.
GCD(x,y)=GCD(x DIV 2,y) if x is even and y is odd
GCD(x,y)=GCD(x-y,y) if x and y are both odd and x>y.

Ex. 13 Suppose that we make the GCD procedure of Exercise 8 into an infix operator .GCD and then evaluate .GCD/s for a set s. What result does this produce?. Assuming that s1 and s2 are non-null sets is the identity, is

```
GCD/(s1+s2)=(.GCD/s1).GCD/s2
```

always true? What will happen if one of s1 or s2 is null?

Ex. 14 A fraction m/n (with integer numerator and denominator) can be represented in SETL as an ordered pair [m,n]. Using this representation, write definitions for OPs called .RS, .RD, .RP, and .RQ, which respectively form the sum, difference, product, and quotient of two fractions. These operators should reduce fractions to lowest terms, for which purpose one of the GCD procedures developed in Exercise 12 will be found useful.

Ex. 15 Supposing that fractions have the representation described in Exercise 14, write a procedure which takes a set of fractions and sorts them into increasing numerical order.

Ex. 16 The following mathematical relationships can be used as the basis for recursive procedures for calculating various functions . Write out appropriate recursive procedures for each of these functions.

(a) The value x occurs as a component of a tuple t if and only if it occurs either as a component of the left half of t or as a component of the right half of t.

(b) The sum of all the components of a tuple t of intgers is the sum of the left half of t plus the sum of the right half of t.

(c) The reverse of a tuple t is the reverse of its right half, followed by the reverse of its left half.

Think of at least four other relationships of this kind, and write out recursive procedures based on these relationships.

Ex. 17 The Fibonnacci numbers F(n) are defined by the following facts: F(1)=F(2)=1, F(n+1)=F(n) + F(n-1) for n>1.

(a) Write a recursive procedure for calculating F(n).

(b) Write a procedure which calculates F(n) without using recursion.

Ex. 18 Write a recursive procedure to calculate the number of different ways that an integer n can be written as the sum of two squares, as the sum of two cubes, and as the sum of three cubes. Print out a table of these values and see if they suggest any interesting general results.

Ex. 19 To compute the power x**n, one can multiply x**m by x**k for any positive integers m and k satisfying m+k=n. Write a recursive procedure which uses this fact to determine the minimum number M(n) of multiplications needed to calculate x**n. Print out a table of M(n) for all n from 1 to 100. Use the technique explained in Exercise XXX to ensure that your recursive procedure is not unnecessarily inefficient.

Ex. 20 Take Mergesort (Section 4.4.2) and one other recursive procedure, and track their recursive operation by inserting code which computes the level of recursion reached by every invocation of the procedure being tracked. (A global variable should be introduced for this purpose). Messages like the following should be printed:

```
invoking Mergesort from recursion level 3
entering Mergesort at recursion level 4, parameter is...
returning from Mergesort to recursion level 3, result is...
```

Ex. 21 The correlation corr(u,v) of two vectors u,v of n real numbers is the quotient

(u(1)-Mu)*(v(1)-Mv)+...+(u(n)-Mu)*(v(n)-Mv)/sqrt(VAu*Vav)

where Mu and Mv are the means of u an v respectively, while VAu and VAv are the variances of u and v respectively (see Exercise 6). Write SETL procedures which calculate and return this value. Use this procedure to calculate and print the correlation of ten randomly selected pairs of vectors. What is the largest value that corr(u,v) can possibly have? What is the smallest?

Ex. 22 Write a procedure which will read a number written in any specified number base from 2 to 36, and convert it to the integer it represents in decimal notation. Numbers in bases below ten will involve only the digits '0' thru '9'; numbers written in larger bases will use the capital letters 'A' thru 'Z', in increasing order, as additional digits. For example, base 16 numbers will be written using the characters

0 1 2 3 4 5 6 7 8 9 A B C D E F,

and base 18 numbers will be written using the characters

0 1 2 3 4 5 6 7 8 9 A B C D E F G H I.

Also, write a procedure which will convert an integer to its string representation in any of these bases. These programs should allow for the fact that an illegal character might occur in a string which is to be converted to an integer.

Ex. 23 Write a program which can be used to prepare an alphabetized directory of your friends' names, addresses, and telephone numbers. The input to this program is assumed to be a list of multi-line entries, each starting with a line having the format

*key,

where -key- designates an alphabetic key which determines the alphabetic position of the given entry. (These keys are not to be printed in the final directory.) For example, two entries might be

```
*Smith
Mary Smith
222 Flowery Way
Ossining, N.Y.  10520
*Termites
Acme Exterminators
(Termite Specialists)
(Recommended by Mary)
(202)789-1212
```

Ex. 24 Write a 'personalized letter' generator. The inputs to this program should be a form letter L and a file F containing 'addresses' and 'variations'. The letter L is given as a text containing substrings **j**, and the file F given as a sequence of items **s1**s2**...**sn, each sj being some 'personalising' string. The expanded form of the letter is produced by inserting the address in an appropriate position, and replacing each

substring **j** in the form L by the string sj. For example, if L begins

```
 Dear **1**:
Since only **2** weeks reman before you will graduate from
**3**,
...
```

and the first entry in F is

```
 Ms. Nancy Holman
353 Bleeker St
N.Y.C., 10012 NY
**Nancy**six**New York University
```

the 'personalized' letter generated will be

```
 Ms. Nancy Holman
353 Bleeker St
N.Y.C., 10012 NY

 Dear Nancy:
 Since only six weeks remain before you will graduate from
New York University,...
```

The personalized letters that your program generates should be right-justified and attractively formatted. Try to think of, and implement, features which will improve the utility of the personalized letter generator.

Ex. 25 Write a procedure which will print a string of up to six characters in 'banner' format on your output listing. In this format, each character is printed one and a half inches wide and two inches high; the whole banner should also be centered on the listing.

Ex. 26 The set of distances between the centers of cities x,y directly connected by a road not going through any other city is given by a map dist(x,y). (Whenever dist(x,y) is defined, so is dist(y,x), and of course dist(x,y)=dist(y,x).) Write a program that will use this information to calculate the shortest driving distance between any two cities (whether or not they are connected directly by a road). This information should be printed out as an inter-city distance chart of the usual form. Also, print out a chart which describes the shortest driving route between cities by listing the city z that you should drive to first if you want to go from x to z.

Ex. 27 Write a procedure which, given two tuples t1 and t2, prints out a list of the number of times each component of t1 occurs as a component of t2.

Ex. 28 Write a procedure whose parameters are a string x and a set s of strings, and which returns the element of s which has the largest number of sucessive character pairs in common with x. How would you structure this procedure if it is to be called repeatedly, always with the same s, but with

many different values of x?

Ex. 29 Write a code fragment that determines whether a character C is a letter, digit, blank, or special character. Try to make your code efficient.

Ex. 30 Manhattan island was purchased in 1626 for $24. If instead this money had been deposited in a bank account drawing 6% annual interest, how much would it be worth now?

4.7 Other PROCEDURE-related facilities

4.7.1 Procedures and functions with a variable number of arguments

Occasionally one wants to write a procedure or function which can accept a variable number of arguments. One may, for example, want to write a function which sums the value of all its integer arguments, or a procedure which can take any number of arguments, capitalize, and print them. Another example is furnished by SETL's built-in -read- and -print- functions; the -print- function accepts any number of arguments and prints them one after another, the -read- function accepts any number of arguments and modifies them all by assigning to them SETL values read from input.

SETL does in fact allow such procedures and functions to be written. To define a function with a variable number of parameters, a header- line of the form

```
PROC function_name(x1,x2,...,xn(*))
```

must be used. Here as before, any -function_name- can be used to name the function, and x1,...,xn are as usual its parameters. However, a function declared in this way can be invoked with any number of arguments greater than n-1. All arguments from the n-th onward are then gathered into a tuple which is assigned as the value of the last parameter xn. Thus, for example, in the body of the function, the references xn(1) and xn(5) would refer to the n-th and (n+4)-th argument respectively. Only the last parameter of such a function can be followed by the sign (*) to indicate that it actually represents a list of arguments whose length can vary.

The special reserved symbol NARGS can be used within the body of such a function; its value will be the actual number of arguments with which the function was invoked.

Here, for example, is a modified -print- procedure which accepts any number of arguments and prints them one after another, but which starts a new line whenever it begins printing a set or a tuple, or whenever more than five items have been printed on a single line:

```
PROC nicer_print(x(*));

next := 1 ; $ next item to print
```

```
    (WHILE next<=NARGS)

        IF EXISTS j in [next..NARGS MIN (next + 5)]
                |TYPE x(j) IN {'TUPLE', 'SET'} THEN

            print(x(j)); $ then print the tuple or set on its own line
            next:=j+1 ;
         ELSE
            print_on_line(x(next..next+4)) ;
            next +:= 5 ;

         END IF;
    END WHILE;

    END PROC nicer_print;

    PROC print_on_line(t); $ prints the components of t on one line

    CASE #t OF

(0):  RETURN;              $ nothing to print

(1):  print(t(1)) ;

(2):  print(t(1),t(2)) ;

(3): print(t(1),t(2),t(3)) ;

(4): print(t(1),t(2),t(3),t(4)) ;

(5): print(t(1),t(2),t(3),t(4),t(5)) ;

    END CASE ;

    END PROC print_on_line ;
```

The qualifiers RD,RW,WR can be attached to any of the parameters of a procedure having a variable number of arguments. This is shown in the following example, which gives code for a modified -read- operation which 'echos' all the information that it reads, i.e., copies this information to the standard output file.

```
    PROC echo_read(RW x(*)) ;

    (FOR j IN [1..NARGS])
        read(y) ; print(y) ;
        x(j) := y ;

    END FOR ;

    END PROC echo_read ;
```

To use this procedure, we could for example write

```
echo_read(x,y,z);
```

this would read values into x,y, and z in the normal way, but would also print the information that it read.

4.7.2 User-defined prefix and infix operators

Function names must always be written before their lists of arguments, and these agruments must always be enclosed in parentheses. However, for functions of two arguments, 'infix' notation is generally more convenient; for example, it is more convenient to write

```
a+b
```

than to have to write

```
plus(a,b)
```

and certainly

```
a+b+c+d
```

is more convenient than

```
plus(plus(plus(a,b),c),d).
```

For this reason, SETL allows its user to define two-parameter infix operators (and also one-parameter prefix operators, which however are considerably less useful). The names of such operators must be ordinary SETL identifiers to which the character '.' (period) is prefixed. To introduce such operators, a perfectly ordinary function body followed by a trailer line is used, but the header line introducing the operator is changed to

```
OP .name(a) ;   $ to introduce a prefix operator
```

or

```
OP  .name(a,b) ;  $ to introduce an infix operator
```

Suppose, for example, that we wish to introduce an operator called -.dot- which forms the dot-product of two vectors of equal length, i.e. the sum of the products of their corresponding components. This can be done as follows;

```
OP .dot(u,v) ;

If #u /= #v THEN
        print('tuples of mismatched length',u,v);
        RETURN OM;
ELSE
        RETURN +/[u(i)*v(i) : i IN [1..#v] ;
END IF;

END OP .dot;
```

Once this operator has been defined, we can invoke it simply by writing

```
u .dot v
```

Another example is the useful operator -.c-, which forms the composition of two (possibly multivalued) maps: (See Section 2.7.4 for an explanation of the meaning of map 'composition'.)

```
OP .c(f,g);

RETURN {[x,y]: z=g{x}, q in z, y IN f{q}} ;

END OP .c;
```

User-defined infix operators of this kind can be combined with the token ':=' to form assigning operators (see Section 2.12.1). For example, in the presence of the preceding definition we can write

```
f .c:= g;
```

to abbreviate the common construct :

```
f  := f .c g;
```

Moreover, both built-in and user-defined infix operators can be used to form compound operators. For example, we can use the -.c- operator in the following way to write an operator which forms the n-th power of a map f.

```
OP f .to n;

RETURN IF n=0  THEN { }              $ the identity map
       ELSE .c/[f:i IN [1..n]]       END;

END OP .to;
```

User-defined prefix operators are less useful than user-defined infix operators, since they cannot appear in either of these convenient contexts. However, by defining a function of one parameter as an operator rather than an ordinary PROCEDURE, we save what might otherwise be irritating parentheses. For example, if we define a unary operator minus by writing

```
OP .minus(u);

RETURN [-x: x IN u] ;

END op.minus;
```

Then the negative of a vector u can be formed by writing

```
.minus u
```

If instead of this we made -minus- an ordinary function, we would have to write

```
minus(u)
```

instead.

The arguments of a user-defined infix or prefix operation always carry the implicit qualifier RD, so that attempting to give them either of the qualifications WR or RW is illegal. Attempting to attach the qualifier '(*)' (See Section 4.7.1) to a parameter of an infix or prefix operator is also illegal.

The precedence of any user-defined binary operator is lower than that of any built-in binary operator, with the exception of the following comparators and Boolean operators:

```
= /=  < <= > >= IN NOTIN SUBSET INCS AND OR IMPL
```

Assignments and assigning operators seen from the right also have lower precedence than user defined infix operators. User-defined unary operators have the same precedence as built-in unary operators (See Section 2.13 for details concerning operator precedence). The following examples illustrate these rules: If .op is a user-defined binary operator, then

```
a+b .op c  means (a+b) .op c

b .op c = d  means (b .op c) = d

b .op c AND d means (b .op c) AND d

b .op c + d means b .op (c+d)

a +:= b .op c means b .op (c+d)

a +:= b .op c means a +:= (b .op c)
```

4.7.3 Refinements

Procedures play various roles, and in particular serve to clarify the logical structure of a complex program by dividing it into subsections whose names hint at their purposes. However, the use of procedures is a bit 'heavy' syntactically, in part because procedures require header and trailer lines to introduce them, in part because the variables of a procedure are logically isolated from all other procedures. (Unless these variables are made global; but then they become accessible to all procedures, which, as pointed out in Section, 7.1 is often highly undesirable.) This slight clumsiness discourages the use of small groups of short procedures which need to share many variables amongst themselves. To fill the need for a facility of this kind, whose use can aid considerably in documenting and clarifying the logical structure of a program, SETL provides a less powerful but easier-to-use alternative to PROCEDUREs, namely refinements.

A refinement is a block of statements which is labeled by an identifier followed by a double colon, as in

```
solve_equation:: x := (-b+sqrt(b*b-4.0*a*c))/(2.0*a);
```

Within a procedure or a main program block, a refinement can be invoked by using its label as a statement. This is shown in the following example

```
PROGRAM quadratic;

    input_data;  $ this and the next 2 lines invoke refinements

    solve_equation;
    output_results;

solve_equation::  $ a first 'refinement'

    x :=  (-b +sqrt(b*b-4.0*a*c))/2*a;

output_results::    $ a second 'refinement'

    print('Root is',x);

input_data::  $ a third 'refinement'

    read(a,b,c);
    print(a,b,c);
    check_eof;   $ this invokes the fourth refinement shown just below

check_eof::    $ a fourth 'refinement'

    IF EOF THEN print('improper data'); STOP; END IF;

END PROGRAM quadratic;
```

This example illustrates the following rules:

(a) All refinements (if any) must follow at the end of the procedure or main program block within which they are used.

(b) Refinements are written one after another, but can appear in any sequence.

(c) A refinement can be invoked anywhere in a procedure or a main program, but can be invoked only once. If a section of code is to be invoked more than once, it should be made a procedure rather than a refinement.

(d) Refinements have no parameters. They make use of the same variables as the main program block or procedure P to which they belong. Variables used in refinements have the same meaning that they would have in (this block or procedure) P. Refinements are executed by inserting the series of statements of the refinement in place of the reference to the refinement.

4.8 Rules of Style in the Use of Procedures

Effective programming depends more on the proper use of procedures than on any other single factor. Your use of procedures should aim to achieve various important stylistic goals:

(a) Procedures are used to 'paragraph' programs, i.e., to divide them into manageably short subsections, each performing some easily definable logical function, which can be read and understood in relative independence from each other. Here the key term is independence: it is important to write your procedures in a manner which isolates each of them as much as possible from the internal details of other procedures. Only a small number of well-defined data objects should be passed between procedures. Very few data objects should be shared globally between procedures; sharing is dangerously productive of errors, so that all data object sharing should be carefully planned, should adhere to clearly understood stylistic rules, and must be scrupulously documented. Be sparing in your use of global VAR declarations!

(b) Procedures are also used to abbreviate, i.e., to give frequently used compound constructions a name facilitating their repeated use. This usage will often give rise to short procedures, the shortest of which may reduce to a single RETURN statement. Code sequences used more than a very few times should be replaced by short procedures, since such procedures will only need to debugged once, while repeated code sequences can be repeated incorrectly, and can interact in unanticipated ways with code surrounding them (for example, by accidental overlap of names). These facts make repetition of code sequences dangerous, and their replacement by procedures advantageous.

(c) Procedures define one's conceptual approach to a programming task, and are used to clarify and help document programs. If this is done well, a program's topmost procedure will document the main phases of the program and explain the principal data structures passed between its phases. Then each intermediate level procedure will both realize and 'flow chart' an important substep of processing. Each bottom-level procedure will realize some well-defined utility operation and will be separately readable. The narrative commentary which accompanies the program should be organized around the layout of its procedures. Comments concerning overall approach and main shared data objects will accompany top-level procedures, and detailed remarks on particular algorithms will be attached to the low-level subprocedures which implement these algorithms.

(d) Procedures are used to decompose programs into separate parts which have different degrees of generality/specificity, or which have significantly different 'flavors' in some other regard. The 'buckets and well' example considered in Section 4.3.1 exemplifies this point. In this program, procedures new_states_from, pour, fill, etc. concentrate all details particular to the specific problem being solved, while procedure find_path, which simply realize a general technique for searching overstates and constructing paths are independent of these details. This separation makes it possible to use find_path to solve other problems of the same kind, simply by replacing new_states_from and pour, etc.

(e) When one is writing a program which addresses a mathematical or application area which makes use of some well-established set of concepts, it can be very advantageous to define SETL representations for all the kinds of objects used in this area, and then to write a collection of utility procedures which can be used to apply all the important operations of the area to these objects. These procedures should be written in a way which allows their user to ignore the internal details of the object representations, making it possible for him to think more as a specialist in the application area rather than as a programmer. This is the important principle of 'information hiding': structure your programs in a way which allows the representational details of objects manipulated by the highest level programs to be concealed from the authors of these programs. (So important is this principle that some modern programming languages include syntactic mechanisms for enforcing it rigorously.) A family of procedures which manipulate objects whose internal representational details are known only to these procedures is sometimes called a package. The package of polynomial manipulation procedures shown in Section 8.6.5 is an example; other examples appear in the exercises listed in Section YYY.

It is worth saying a bit more concerning the paragraphing of code. Elusive errors easily creep into codes whose logic is spread over very many lines. For this reason, one should always strive to break codes into independent 'paragraphs' no more than ten or so lines in length. (Longer paragraphs can be used where this is unavoidable, but as these grow to a page or more in size, the likelihood of troublesome multiple errors, as well as the difficulty of understanding what is going on when the code is read subsequently, will rise rapidly.) The three main constructs which can help you to paragraph code adequately are

(i) use of procedures and functions
(ii) use of refinements (see section 4.7.3)
(iii) use of the CASE statement

Each procedure, function, and refinement whose integrity is not compromised by an undisciplined use of shared global variables constitutes an independent paragraph of code. Moreover, since only one of its alternatives will be performed each time a CASE statement is executed, the seperate alternatives of a CASE statement can be regarded as independent paragraphs. Hence, whenever the body of a procedure extends over more than a few dozen lines, most of this body should consist of one or more CASE statements each of whose alternatives is short. If this is not done, then the rules of good style are being violated; and this violation should either have compelling justification or be removed.

Nesting of loops and of IFs also raises interesting stylistic questions. Since iterations will rarely be nested more than three deep, nested iterations can generally be used without significant confusion resulting. When deeper nests start to build up, or even the body of an outermost iteration tends to grow long, an effort should be made to relegate parts of its body to one or more separated subprocedures.

Deep nesting of IFs leads very rapidly to confusion. Where at all possible nested IF's more than two deep should be replaced by uses of CASE statements, or by segregation of the more deeply nested alternatives into procedures. A third alternative is to 'flatten' a deeply nested IF

construct into an IF-construct which is less deeply nested, but in which the alternatives of the original IF-nest have been combined using the Boolean AND, OR, etc. (However, this will tend to generate longish sequences of ELSEIF's.) For example, instead of writing

```
IF a>0 THEN
   IF b<0 THEN
      a +:= 1 ;
   ELSE
      a -:= 1 ;
   END IF ;
ELSE
   IF b<0 THEN
      b +:= 1 ;
   ELSE
      b -:= 1 ;
   END IF ;
END IF ;
```

it is preferable to 'flatten' and write

```
IF a>0 AND b<0 THEN
            a +:= 1 ;
ELSEIF a>0 AND b>=0 THEN
            a -:= 1 ;
ELSEIF a<=0 AND b<0 THEN
            b +:= 1 ;
ELSEIF a<=0 AND b>=0 THEN
            b -:= 1 ;
END IF ;
```

Still better, one can use the following CASE statement:

```
    CASE OF
(a>0 AND b<0): a +:= 1 ;
(a>0 AND b>0): a -:= 1 ;
(a<0 AND b<0): b +:= 1 ;
(a<0 AND b>0): b -:= 1 ;
    END CASE ;
```

Note than an extended the IF..ELSEIF..ELSEIF...construct has some of the same paragraphing advantages as an extended sequence of CASE alternatives. However, IF alternatives are less fully independent than CASE alternatives, since implicit conditions accumulate from each branch of an IF statement to the next. Some of the confusion which this will cause can be avoided by using auxiliary comments to indicate the conditions under which each branch of an extended IF will be executed, but is is even safer to use a CASE statement instead.

4.9 Exercises

The 'dot-product' of a pair u,v of equally long vectors with integer or real coefficients is the sum +/[u(i)*v(i):i IN [1..#v]].

Ex. 1 Write a SETL OP definition for an infix operator .DP such that x .DP y is the dot-product of the vectors x and y. Write a prefix operator .RV n which returns a randomly chosen integer-valued vector of length n each time is is invoked. Use these two functions to test the validity of the following statements concerning vector dot products:

```
(a)    (x .DP y)=(y .DP x)
(b)    (x .DP x) >= (MAX/x)*(MAX/x)
(c)    (x .DP y)**2 <= (x .DP x)*(y .DP y)
(d)    (x .DP y) <= (MAX/x)*(MAX/y)*#x
```

Ex. 2 The sum of two integer or real vectors x, y of equal length is [x(i)+y(i):i IN [1..#x]], and their difference is [x(i)-y(i):i IN [1..#x]]. Write definitions for two OPs called .S and .D which produce these two vectors. Proceed as in Exercise 1 to test the following statements:

```
(a)    ((x .S y) .S z) = (x .S (y .S z))
(b)    (x .S (y .D x)) = y
(c)    ((x .S y) .DP z) = (x .DP z) + (y .DP z)
(d)    ((x .D y) .DP z) = (x .DP z) .D (y .DP z)
```

Ex. 3 Write a procedure which, given two tuples t1 and t2, prints out a list of the number of times each component of t1 occurs as a component of t2. Write another procedure which, given a tuple t, calculates a map which sends each component x of t into the index of the first occurence of x within t. Express f in terms of t, in the simplest way you can.

Ex. 4 The storage space needed to represent a map f can sometimes be reduced very considerably by writing f in the form f(x) = f1(x)? (IF x IN s THEN f2(x) ELSE OM END), where f1 has a small domain, s has a simple representation, and f2 is a programmed function. Write a procedure -compress- which, given f, s, and f2, will calculate f1. The function f2 should be called by -compress-, and it is assumed that user of the -compress- is required to supply code representing f2.

Ex. 5 Write a room assignment program which reads information concerning available rooms and classes needing rooms, and generates a room assigment. The first of the two data items read by your program should be a map from room numbers to seating capacities. The second input read by your program should be a tuple of triples, each consisting of a class number (a string of the form n.m where n is a course number and m a section number), number of students, and hour (Possible hours are 8,9,10,11,.. up to 20). No two classes meeting at the same hour should be scheduled into the same room. Your program should print out a table, arranged by hour and room, of assignments. Starting with the largest class scheduled to meet in a given hour, each class should be assigned the smallest room into which it will fit. Classes which cannot be scheduled should be appropriately listed. Empty rooms should be indicated in the output table you print.

The next three exercises relate to the earlier exercises on Boolean identities, found in Section 2.3.4.1.

Ex. 6 Boolean 'implication', which we will write as an infix operator x .IMP y is TRUE if either x is FALSE or y is TRUE. Thus x .IMP y is equivalent to (NOT x) or y. Write a SETL op definition for this operator, which will be used in the next two exercises.

Ex. 7 Using the .IMP operator defined in Exercise 1 and the method for checking Boolean statements described in Section 2.3.4.1, show that each of the following statements is true no matter what the Boolean values of the variables occuring in it.

```
(a)     (x OR NOT y)=(y .IMP x)
(b)     ((x AND y) .IMP z)=(x .IMP (y .IMP z))
(c)     (x .IMP (y OR z))=((x .IMP y) OR (x .IMP z))
(d)     ((x .IMP y) AND x) .IMP y
(e)     (x .IMP NOT x) .IMP NOT x
(f)     x .IMP (y .IMP x)
(g)     (NOT x) .IMP (x .IMP y)
```

Ex. 8 None of the following Boolean formulae are valid for all Boolean values of x and y; each represents a common logical fallacy. Proceeding as in Exercise 2, write a SETL program which will find a case in which each of these formulae evaluates to FALSE.

```
(a)     ((x .IMP y) AND y) .IMP x
(b)     ((x .IMP y) AND (x .IMP z)) .IMP (y .IMP z)
(c)     ((x OR y) AND x) .IMP NOT y
(d)     ((x .IMP y) AND NOT x) .IMP NOT y
```

Ex. 9 When a sequence of data items are read by a read statement of the form

```
read(x,y,..z),
```

it will often be appropriate to check the items read to make sure that they have appropriate types and lie in appropriate ranges. For this purpose, the following approach, based upon the notion of a 'descriptor string', is may be convenient:

(a) Capital letters are used in the following way to designate the principal SETL object classes:

letter	value	letter	value
I	integer	T	tuple
R	real	E	set
S	string	A	atom

(b) The ranges of integers and of real numbers can be constrained. For example, I-100..100 designates an integer belonging to the set {-100..100}, I0.. designates a non-negative integer, R-1.0..1.0 designates a real number lying between -1.0 and +1.0.

(c) The descriptors T and E can be qualified to show the types of their components or members. For example T(IIR) describes a tuple of length 3 whose components are an integer, an integer, and a real respectively; T.I

describes an unknown-length tuple of integers; E.T(II) describes a set of pairs of integers.

(d) To describe successive items in a list of variables being read, descriptors are simply concatenated. For example, if three items x,y,z, the first an integer, the second a set of pairs of integers, and the third a tuple of strings, are being read, we would describe it by IE.T(II)T.S.

Write a multi-parameter procedure read_check whose first parameter is a descriptor string defining the data expected and whose remaining parameters are the variables whose values are to be read. E.g., in the example appearing in (d), we would write

```
read_check('IE.T(II)T.S',x,y,z);
```

The read_check procedure should generate a report if data it encounters any of data unexpected form. Of course, the read_check procedure must be foolproof.

Ex. 10 Extend the read_check procedure of Exercise 9 so that any data item to which there corresponds a descriptor followed by the letter 9 will be checked for membership in a set of possible values that is given explicitly. This set should come directly after the data-item being read, in the list of arguments of the extended read-check procedure. For example, if we expect x to be an indication ('M' or 'F') of sex, and y to be an age, we could write

```
read_check(SXI0..150,x,{'M','F'},y);
```

Ex. 11 Modify the read_check procedure of Exercise 8.6.5 so that it echoes and labels all data read. For this modified procedure, the sequence of names of the variables being read should follow the data descriptor in the procedure's first parameter. These names should be separated from the data descriptor and from each other by blanks.

$

CHAPTER 5

DATA OBJECTS AND EXPRESSIONS, CONCLUDED

In this chapter we will complete our discussion of the various classes of data objects supported by SETL, and of the forms of expression which the language provides.

Chapter Table of Contents:

5.1 Real Operators:

+, -, *, /, **, =, >, <, >=, <=, MAX, MIN, ATAN2, ABS, FIX, FLOOR, CEIL, ACOS, ASIN, ATAN, COS, EXP, LOG, RANDOM, SIGN, SIN, SQRT, TAN, TANH

Binary real operators compute a result value from two real values, x and y. The binary real operators provided by SETL are as follows:

x+y	computes the sum of x and y.
x-y	computes the difference of x and y.
x*y	computes the product of x and y.
x/y	computes x divided by y. An error results if y is zero, of if the division causes floating point overflow.
x**i	this variant of the exponentiation operator yields x raised to the integer i. An error results if exponentiation causes floating point overflow, or if x and i are both zero.

x=y	yields TRUE if x and y are equal, FALSE otherwise.
x/=y	yields TRUE if x and y are unequal, FALSE otherwise.
x>y	yields TRUE if x is greater than y, FALSE otherwise.
x<y	same as y>x.
x>=y	yields TRUE if x is at least as large as y, FALSE otherwise.
x<=y	same as y >= x.
x MAX y	yields the larger of x and y.
x MIN y	yields the smaller of x and y.
x ATAN2	yields the arc tangent of the quotient x/y. The result is given in radians.

Unary real operators compute a result value from a single floating point input x. The unary real operators are as follows:

+ x	yields x.
- x	yields the negative of x.
ABS x	yields the absolute value of x, i.e. yields x if x is positive, -x if x is negative.
FIX x	yields the integer part of x, dropping its fractional part.
FLOAT i	yields a real quantity numerically equal to i, where i is an integer.
FLOOR x	yields the largest integer which is not larger than x. (See the examples given below for the rules which applies if x is negative).
CEIL x	yields the smallest integer which is at least as large as x. (See the examples given below for the rule which applies if x is negative).
EXP x	yields e**x, where e is the base of natural logarithms.
LOG x	yields the natural ('base e') logarithm of x. An error results if x is zero or negative.
COS x	yields the cosine of x, which is assumed to be given in radians.
SIN x	yields the sine of x, which is assumed to be given in radians.
TAN x	yields the tangent of x, which is assumed to be given in radians.
ACOS x	yields the arc cosine of x; the result is given in radians. An error results if x does not lie between -1.0 and +1.0.
ASIN x	yields the arc sine of x; the result is given in radians. An error results if x does not lie between -1.0 and +1.0.
ATAN x	yields the arc tangent of x; the result is given in radians.
TANH x	yields the hyperbolic tangent of x.
SQRT x	yields the square root of x. An error results if x is negative.
RANDOM x	yields a real value which is randomly distributed over the range from zero to x including zero but excluding x. Note that successive calls to this function will return distinct, independently choosen random quantities.
SIGN x	yields one of the integer results -1, 0, or +1 depending on whether x is negative, zero, or positive.

Examples of some of these operators are as follows:

1.1 + -1.1	yields 0.0
1.1 * 1.1	yields 1.21
1.1 ** 2	yields 1.21
1.1 ** 2.0	yields 1.21

```
1.1 = 1.11          yields FALSE
1.1 = 1.10          yields TRUE
1.1 MAX 1.1001      yields 1.1001
1.1 MIN 1.101       yields 1.1
+1.1                yields 1.1
- -1.1              yields 1.1
ABS -1.1            yields 1.1
print(FIX 1.1, FIX -1.1)                      yields  1, -1

print(FLOOR 1.1, FLOOR -1.1, FLOOR -1.0)      yields  1, -2, -1
print(CEIL 1.1, CEIL -1.1, CEIL 1.0)          yields  2, -1,  1

print(FLOAT 1, FLOAT -1, FLOAT 0)             yields  1.0, -1.0, 0.0
print(FLOAT 1234567891234567891234567891234567891234567889)
                  can result in an overflow error.
```

The forms in which real constants can be written are described in Section 2.1.1.

Note that for real numbers x and y, the use of the comparators x=y and x/=y can be a bit tricky since rounding effects might cause (0.5 + 0.5) = 1.0 to yield FALSE and 1.0 = 1.0000000000000000001 to yield TRUE. Keep in mind the fact that real values can always turn out to have values slightly different from the exact values that you may expect.

5.2 String Scanning Primitives

SETL supports some of the handy string-string primitives whose use was pioneered by the designers of the SNOBOL programming language. These generally have the form

(1) operation_name(scanned_string,pattern_string).

Each of these operations attempts to match a portion of its scanned_string parameter in a manner defined by the pattern_string. If a portion of the scanned_string is successively matched, it is removed from the scanned_string and becomes the value of the function (1). If no portion of the scanned_string is matched by (1), then scanned_string is not changed, and the value of the function (1) is OM. Since these operations write their first parameter, only expressions which can appear on the left hand side of an assignment are acceptable in the scanned_string position of (1).

The first of these scanning operation: namely,

(2) SPAN(ss,ps)

scans over as large an initial part of ss as consists of characters which belong to ps. This part of ss is broken off, and becomes the value of the function (2); the remainder becomes the new value of ss. If not even the first character of ss belongs to ps, then ss is unchanged and the function (2) yields OM.

Here are a few illustrations of the action of the SPAN primitive: Suppose that ss has the value 'If, gentlemen'. Then

```
ANY(ss,'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghij')
```

has the value 'If' and gives ss the value ', gentlemen'. Also,

```
SPAN(ss,'abcdefghijklmnopqrstuvwxyz')
```

has the value OM and does not change the value of ss.

The remaining string-scanning primitives provided by SETL are as follows:

(2) ANY(ss,ps)

breaks off and yields the first character of ss if this belongs to ps. If the first character of ss does not belong to ps, then ss is unchanged and the value returned by ANY is OM. For example, the code fragment

```
ss := 'ABC'
print(ss,ANY(ss,'AEIOU'),ss,ANY(ss,'AEIOU'),ss);
```

will yield

```
ABC   A   BC   OM   BC .
```

The scanning primitive

(3) BREAK(ss,ps)

scans ss from the left up to but not including the first character which

does not belong to ps. This part of ss is broken off and becomes the value of the function (3). If ss contains no characters not belonging to ps, then (3) has the value OM and ss is not changed. If the very first character of ss belongs to ps, then (3) has a nullstring value and ss is not changed.

The scanning primitive

(4) LEN(ss,n)

has an integer second parameter. If #ss >= n, then (4) yields the value ss(1..n) and the assignment ss := ss(n+1..) is performed; otherwise (4) yields OM and ss is not changed. The primitive

(5) MATCH(ss,ps)

yields ps if #ps <= #ss and if ps = ss(1..#ps), then the assignment ss := ss(#ps+1..) is performed. Otherwise (5) yields OM and ss is unchanged. The primitive

(6) NOTANY(ss,ps)

breaks off and yields the first character of ss that does not belong to the string ps. In the contrary case (6) yields OM and ss is unchanged.

Each of the above string primitives is also provided a 'right-to-left' form which starts from the right, at the last character of the scanned string, rather than from the left to right, starting at the first character of the scanned string as in the cases already considered. The following table shows the right-to-left variant of each of the primitives described above.

Left-to-right variant	Right-to-left variant
ANY(ss,ps)	RANY(ss,ps)
BREAK(ss,ps)	RBREAK(ss,ps)
LEN(ss,n)	RLEN(ss,n)
MATCH(ss,ps)	RMATCH(ss,ps)
NOTANY(ss,ps)	RNOTANY(ss,ps)
SPAN(ss,ps)	RSPAN(ss,ps)

Two additional string utilities are provided to make productions of decently formatted string output easier. These are

LPAD(ss,n) and RPAD(ss,n)

The LPAD primitive returns the string obtained by padding its first argument ss out to length n (which must be an integer) by adding as many blanks to the left of ss as necessary. If #ss=n, then LPAD(ss,n) is simply ss. The RPAD primitive behaves similarly, but adds blanks on the right.

5.2.1 Examples of Use of the String Scanning Primitives

5.2.1.1 A Simple 'Lexical Scanner'

One of the first problems that arises when one begins to program a compiler for a programming language (like SETL, BASIC, or any ofthe other language with which you may be familiar) is to break the original or 'source form' of the pogram into a stream of individual identifiers, constants and operators (collectively, these items ae called 'tokens'). The program, which the computer will read, must be decomposed into these elements before we can determine its meaning. For example, on reading the fragment

'A0 = B1 * C1 + 3.78'

of text, one must break it up into the tuple

['A0', '=', 'B1', '*', 'C1', '+', '3.78'].

Note that the first of these items is an identifier, the second an operator sign, the last a constant, etc. (Blanks separating tokens are ordinarily eliminated as the source text is scanned).

A procedure which performs this kind of decomposition of strings representing successive lines of program text is called a lexical scanner.

It is easy to write a lexical scanner for a simple language using the string scanning operations that we have just described. We will now show how to do this, but to avoid complications, we will suppose that the following rules apply:

(a) The program text to be scanned contains only identifiers, operator signs, integers, real constants and blanks.

(b) An identifier is any string starting with an alphabetic and containing only alphabetic and numeric characters.

(c) Any 'special' character (i.e. characters like '+', '-', ',' and ':', which are not blank, alphabetic, or numeric) will be regarded as an operator.

(d) An integer is a sequence of numerics not followed by a period. A real number is a string of numerics including at most one period.

From the string being analyzed, the following procedure repeatedly breaks off a section consisting of a run of blanks, a run of digits, an identifier or a single 'special' character of some other kind. Blanks are ignored. If a run of digits is found, we check to see if a decimal point and a second run of digits follow. If so, they are concatenated to the run of digits originally found. In each case, a nonblank section broken from ss constitutes a token, and it is added to the tuple of tokens which is eventually returned. The code assumes that -num- and -alphanum- are constants which must be initialized as follows:

```
num := '0123456789'
alphanum := 'abcdefghijklmnopqrstuvwxyz'
            + 'ABCDEFGHIJKLMNOPQRSTUVWXYZ' + '0123456789';
PROC lex_scan(stg);          $ lexical scan routine where the
                             $ parameter is a string.
tup := [ ];                  $ Initialize the tuple to be returned.
stg +:= ' ';                 $ Add a terminating blank.
(WHILE stg /=' ')
```

```
        token := SPAN(stg,' ')? SPAN(stg,num)? SPAN(stg,alphanum)?
                 LEN(str,1);
                                  $ Break off a run of blanks, a number,
                                  $ a variable name or a single letter.
        IF token(1) = ' ' THEN CONTINUE; END;    $ Ignore blanks.
        IF token(1) IN num THEN  $ Test for following '.' and
                                  $ numerics.
          IF MATCH(stg,'.') THEN $ Look for digits following
                                  $ the decimal point.
             token +:= '.' + (SPAN(stg,num)?'');
          END IF MATCH;
        END IF token;
        tup +:= token;  $ Add token to tuple being built up.
        END WHILE;
      RETURN tup;
      END PROC lex_scan;
```

5.2.1.2 A 'Concordance' Program

The following code generates a 'cross reference listing' or 'concordance' of a source text. The source text is assumed to consist of a sequence of strings containing words separated by punctuation marks or blanks. The words present in the source text are printed in alphabetical order, each word being followed by a formatted list of all the lines in which the word occurs.

```
PROGRAM concordance;     $ concordance generator

VAR capital_of;          $ All upper and lower case alphabetics.
initialize(capital_of, alphabetics);

line_number := 0;        $ Initialize line_number count.
lines_word_is_in :={};   $ Initialize this to the empty map.

(WHILE (tuple_of_words := break_next_line(line_number)) /= OM)
      $ break_next_line is assumed to read a line of text
      $ and to decompose it into the words it contains by capitalizing
      $ them and eliminating punctuation marks.
      (FOR word IN tuple_of_words)
         lines_word_is_in(word) :=
                  lines_word_is_in(word)?[] WITH line_number;
      END FOR;
END WHILE;

$ Now sort, putting all words encountered into alphabetical order.
$ This is done using the procedure described in Section 4.4.1.

(FOR [word,lines] IN
         sort({[wd,[wd,lns]]:[wd,lns] IN lines_word_is_in}))
      print(word); arrange(lines);     $ Arrange the line numbers neatly.
END FOR;
```

```
PROC break_next_line(RW line_number);  $ Input and scanning routine.
$ This procedure reads a line of input and scans it
$ to break out the words which it contains.  These words are
$ capitalized and placed in a tuple.

line_number +:= 1;          $ Advance the line number.
read(line);
IF EOF THEN RETURN OM; END;

words := [];                $ Start a new tuple of words.

(while line /= '')          $ Until the line has been digested.
 break(line,alphabetics); $ Drop any leading nonalphabetic characters.
  words with := capitalize(SPAN(line,alphabetics));
        $ Note that the SPAN is OM if the line is empty.
END WHILE;

RETURN words;

END PROC break_next_line;

PROC arrange(lines);      $ Routine to print sequence of line numbers.
$ This routine prints up to ten line numbers per line of the concordance
$ and arranges them neatly in fields six characters wide.

(WHILE lines /= [])         $ Until all line numbers are processed,
  group := lines(1..10); $ break off a first group of up to ten lines.
  lines := lines(11 MIN (#lines+1)..);
  print(12*' '+/[LPAD(STR ln,6): ln in group ]);
END WHILE;

END PROC arrange;

PROC capitalize(word);        $Capitalizes its parameter.

RETURN IF word = OM THEN OM  $ Returning capitalized version.
   ELSE ''+/[capital_of(let)?let]: let IN word] END;

END PROC capitalize;

$ The -sort- procedure which should appear here is the one
$ shown in Section 4.4.1; we do not repeat it.

PROC initialize(RW capital_map, RW alphabet_string);
     $ Initialization routine.

small_lets := 'abcdefghijklmnopqrstuvwxyz';
big_lets := 'ABCDEFGHIJKLMNOPQRSTUVWXYZ';
alphabet_string := small_lets + big_lets;
capital_map := {[small_let,big_lets(i)]: small_let=small_lets(i)};

END PROC initialize;

END PROGRAM concordance;
```

5.2.1.3 A 'Margin Justification' Procedure

Our third example is a 'margin justification' procedure which takes a sequence of words separated by blanks, and arranges them into lines which fit between left_margin and right_margin with the first nonblank character placed in position left_margin and the last nonblank character placed in position right_margin. Extra blanks are inserted at random positions between the words to force 'justification' of the right margin. Procedures of this sort are often used in text preparation programs.

```
PROC justify(tuple_of_lines, left_margin,right_margin);

tuple_of_words := [] +/ [break_words(line): line IN
                                    tuple_of_lines];

(UNTIL is_last)

    line_words := break_next_line(tuple_of_words,
                        right_margin - left_margin+1);
    $ break_next_line breaks off and returns the tuple of words
    $ to be placed on the next line.

    IF (is_last := (tuple_of_words = [])) THEN
            $ Output last line with no justification.
        print((left_margin)*' ' +/
            [word+' ': word IN line_words]);
    ELSE    $ Print justified line.
        spaces :=    $ Calculate vector of extra spaces.
          put_spaces(#line_words,right_margin
                    -(left_margin-1+/[#word+1: word IN line_words]));
        print((left_margin-1) * ' ' + line_word(1)
            +/ [line_word(i+1) + (nspace +1) * ' ': nspace=spaces(i)]);
    END IF;

END UNTIL;

END PROC justify;

PROC break_words(line);  $ Breaks line at blanks and returns a tuple
                         $ of words.

tup := [ ];              $ Initialize tuple.

(WHILE line /='')

    IF (word := BREAK(line,' ')) /= OM THEN
        tup WITH := word;
    ELSEIF SPAN(line,' ') = OM THEN  $ last word
        tup WITH := line;
        QUIT;
    END IF;

END WHILE;
```

```
    RETURN tup;

    END PROC break_words;

    PROC break_next_line(RW tuple_of_words, nchars);
    $ This procedure breaks off and returns the longest sequence of words
    $ that will fit into -nchars- character positions;
    $ this sequence is broken off from tuple_of_words.
    sum := 0;

    (FOR word  = tuple_of_words(i..))
        IF (sum +:= #word + 1) > nchars THEN  $ Too far, back up one word.
            save := tuple_of_words(1..i-1);
            tuple_of_words := tuple_of_words(i..);
            RETURN save;
        END IF;

    END FOR;

    save := tuple_of_words;
    tuple_of_words := [];     $ Else this is last line; return all words.

    RETURN save;

    END PROC break_next_line;

    PROC put_spaces(between_kwords, nblanks);
    $ This procedure finds the positions where n blanks are to be placed
    $ between k words.  The blanks are placed at random for
    $ appearance's sake.

    space_count := (size := (between_kwords-1)) * [0];

    (FOR j IN [1..nblanks])
        space_count(RANDOM size) +:= 1;     $ Place a blank.
    END FOR;

    RETURN space_count;

    END PROC put_spaces;
```

Additional procedures related to the above are described in Exercises 14-16.

5.3 Atoms

Mathematical constructions occasionally make use of abstract 'points' which have no particular properties other than their identity. For example, in dealing with graphs we generally regard them as abstract collections of points (or 'nodes') connected by edges (See Figure 5.1).

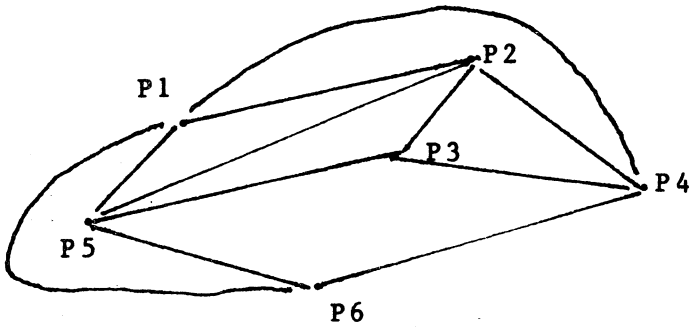


Figure 5.1: A graph: six nodes connected by edges.

In this case, to make a new copy of a graph we need a supply of new 'points'. What these 'points' are is of no significance as long as they can be generated in a way which guarantees that all newly generated 'points' are definitely distinct from all such 'points' previously encountered.

To handle situations of this sort, SETL provides a special kind of object called an 'atom', or for emphasis a 'blank atom'. These objects can be members of sets or components of tuples, but very few other operations act on these atoms. In particular, there is only one way of producing objects of this kind: namely, by calling a special, built-in and argument-free (i.e. 'nullary') function written as

NEWAT

Each time a program invokes this construct, it yields a new atom, distinct from all previously generated atoms. The only operations involving a pair of atoms a and aa, are

```
a = aa    yields TRUE if a and aa are the same, FALSE otherwise
a /= aa   yields TRUE if a and aa are different, FALSE otherwise.
```

In addition, atoms can be made members of sets or tuples (e.g. by the WITH operator) and can be tested for set membership (by the IN and NOTIN operators). Moreover, previously generated atoms which have been put into sets or made into components of tuples can reappear when one iterates over a set of tuple in which they have been placed.

To facilitate debugging of programs which use atoms, the -print- (but not the -read- operation) can be applied to atoms. The internal representation of an atom carries a system-generated integer (not accessible to the SETL user) called its serial number; when an atom is printed, the representation of it is placed on the output medium as

#nnn

where nnn is the serial number of the atom. Thus, for example, if the very first statement in a program is

```
print({NEWAT: j IN [10..20]})
```

the output produced, namely

{#1, #2, #3, #4, #5, #6, #7, #8, #9, #10, #11}

will represent a set of 11 distinct atoms.

Another important use of atoms is to represent objects whcih have a continuing identity, independent of any varying data attributes, associated with them. Consider, for example, the problem of maintaining a simple data base, which keeps track of a few items of data, e.g. name, address, and telephone number for each of a varying group of people.

A given person will of course retain his identity if he (or she) changes his address, telephone number, or even name. Since these informations may change, it is not always appropriate to identify a person with a tuple [name,address,tel_no] even if this tuple gives all available information about him. The most appropriate treatment of such situation may be, in fact, to represent the person by an atom x, and to maintain three maps, called name, address, and tel_no, which map x into the name, address, and telephone number of the person represented by x. Then a name change for person x can be implemented simply by writing

```
name(x) := new_name;
```

To give a small example of the use of atoms, we shall suppose that a graph G is given a a set of ordered pairs, each pair [x,y] representing a directed edge of G going from node x of the graph to node y of the graph. In graph theory, one often wishes to form new graphs from old by introducing new points and edges that serve to simplify some mathematical argument. Suppose, in particualr, that for some reason we wish to introduce two new graph nodes n1 and n2, and to connect n1 to each node of G which is the initial point of an edge in G, and also to introduce an edge [x,n2] for each node x of G which is the second node or 'target' of an edge of G. This will define a new graph G2 within which the original graph G, with all its edges and nodes, is imbedded as a subgraph.

To represent this construction in SETL, it is reasonable to introduce new atoms for the points n1 and n2. This leads us to the following short and quite straightforward code fragment:

```
n1 := NEWAT;          $ Generate first new point.
n2 := NEWAT;          $ Generate second new point.
                      $ Now introduce new edges to build G2.
G2 := G + {[n1,x]: x IN DOMAIN G} + {[y,n2]: y IN RANGE G};
```

5.4 Additional Examples

In this section we collect a few additional examples which illustrate the various use of the operations discussed in this chapter.

5.4.1 Solution of System of Linear Equations

Suppose that we are given a system of n linear equations in n unknowns x1, x2,...,xn. We can suppose that these equations have the form

(1)
```
a11 * x1 + a12 * x2 +...+ a1n * xn = b1
a21 * x1 + a22 * x2 +...+ a2n * xn = b2
an1 * x1 + an2 * x2 +...+ ann * xn = bn.
```

Solution of equations of this kind is one of the most fundamental problems of numerical analysis and has been intensively studied. Without wishing to enter very far into the enormous literature that has developed around this problem, we shall now represent a simple SETL code for solving such systems of equations. The technique we will use is a variant of the famous (though essentially straightforward) technique introduced by Karl Friedrich Gauss (1777-1855), 'The Prince of Mathematicians'. This technique is known as Gaussian elimination.

The idea can be summarized as follows: Each equation in the system (1) involves n coefficients aj1,aj2,...,ajn. If in any equation all of these coefficients are zero, then the whole left-hand side of the equation is zero, and the whole equation reduces to

$$0 = bj.$$

If the quantity bj occurring on the right-hand side is not zero (this is impossible), then the original systems of equations (1) simply has no solutions. At any rate, a system of equations (1) which either contains an equation all of whose coefficients aj1,aj2,...,ajn are zero or whose solution leads to such an equation, is said to be singular. Singular systems of equations require somewhat special analysis. In what follows, we will avoid the analysis and simply assume that the system (1) which we are trying to solve is not singular.

If this is the case, we can take any one of the equations in (1), say the first, and find at least one nonzero coefficient, say alj, on its left-hand side. Then we can pass to an equivalent system of equations by subtracting akj/alj times the first equation from all the k-th equations for each k = 2,...,n. This subtraction eliminates the coefficient akj from all these other equations, i.e. (after subtraction) makes the coefficient akj of the variable xj equal to zero for k = 2,...,n. Hence we can regard equations 2,...,n as a system of (n-1) equations for the (n-1) unknowns, x2,...,xn. Then by proceeding recursively, we can solve these equations for x2,...,xn. Once this has been done, we can substitute the values of x2,...,xn into the first equation, thereby reducing it to a single linear equation in a single unknown. This can then be solved for the remaining variable, x1, by a single subtraction followed, by a division.

We can write a SETL code representing this procedure most clearly if we write it recursively. To do this, we will need to use both an outer procedure -Gauss- which sets up initial parameters and an inner 'workhorse' procedure -Gauss_solve- which performs the actual arithmetic operations. Since the value of the matrix M must be accessed and manipulated by all recursively generated invocations of the -Gauss_solve- routine (see Section 5.4.1), we adopt the (typical) expedient method of making it a global variable. Thus the only parameters that need to be passed to Gauss_solve are a set, namely the set of variables for which a first nonzero coefficient still has to be found and an integer, namely the number of the next equation to be considered. The -Gauss_solve- routine returns OM if it encounters a singularity; otherwise, it returns a vector giving the values of the variables for which it has solved.

```
CONST  eps = 1.0E-4              $ Define a utility constant close to zero.
VAR  glob_M;                     $ Matrix of equation coefficients.
          $ (Note:  these declarations must precede the first PROC).
PROC Gauss(M);                   $ Solves equations by Gaussian elimination.

glob_M := M;                     $ Make  original matrix globally available.
glob_soln := [];                 $ Initialize tuple of solution values.
   RETURN  Gauss_solve({1..#M},1);

END PROC Gauss;

PROC Gauss_solve(var_numbers,next_eqn);
          $ Inner recursion for Gaussian elimination.
          $ var_numbers is the set of all indices of variables
          $ still to be processed; next_eqn is the index of
          $ the next equation to be examined.
IF var_numbers = { } THEN RETURN []; END; $ No variables, return the
                                          $ empty solution.
row := glob_M(next_eqn);                  $ Get the matrix row.
IF NOT EXISTS vn IN var_numbers ST row(vn) > eps THEN
  RETURN OM;                              $ Since system is singular.
END IF;
(FOR j IN [next_eqn+1..#glob_M])
   row_j := glob_M(j);
   subtract := row_j(vn)/row(vn);  $ Multiple of row to be subtracted.
   (FOR vnx IN var_numbers) row_j(vnx) -:= subtract * row(vnx); END;
   M(j) := row_j;
END FOR j;
$ Now call Gauss_solve recursively to solve for the remaining
$ variables.
IF(soln := Gauss_solve(var_numbers LESS:= vn, next_eqn+1)) = OM THEN
   RETURN OM;   $ Since a sigularity has been detected.
END IF;
$ Substitute to determine the value of the vn-th variable.
soln(vn) := (row(#row+1) -/( soln(vnx) * row(vnx): vnx IN var_numbers]))/
            row(vn);
RETURN soln;
END PROC Gauss_solve;
```

It is not difficult to rework this procedure to use iterations rather than recursions. (The iterative form of the procedure is shown below). The relationship between the recursive and iterative forms of code is typical and is worth close study. Note that the iterative form of the procedure must implicitly save information, as the order in which variables are processed, which the recursive form of the procedure saves implicitly (namely in the multiple procedure invocations which are created when the recursive procedure is executed). This is the reason that the quantity -var_order-, which has no counterpart in the recursive procedure, appears in the iterative variant shown below. Aside from this, note that the -Gauss_solve- routine only invokes itself when it is near the point at which it will RETURN; hence the only items of information which need to be saved for use after return from this invocation are -vn- (the number of te variable currently being processed) and -row-. However, -row- is just M(vn); thus only -vn- needs to be saved. This explains why we are able to transform the recursive procedure shown above into the following somewhat more efficient iterative procedure. The initial sequence of recursive calls that would otherwise be required is first represented by a 'forward elimination' pass over the rows of M, and in which the subsequent sequence of recursive returns becomes an iterative 'back-substitution' pass.

```
PROC Gauss(M);              $ Solves linear equations by Gaussian elimination.
  CONST eps = 1.0E-4;       $ Define a constant close to zero.
  soln := [];               $ Initialize solutions to be built.
  var_numbers := {1..n := #M};    $ Initially, all variables need
                                  $ to be processed.
  var_order := [];          $ This tuple will record the order
                            $ in which variables are processed.
    (FOR i IN [1n])         $ Process rows one after another.
       row := M(i);
       IF NOT EXISTS vn IN var_numbers ST row(vn) >= eps THEN
           RETURN OM;       $ Since system is singular.
       END IF;
       (FOR j IN [i+1..n])
            row_j := M(j);
            subtract := row_j(vn)/row(vn);  $ Amount to be subtracted.
            (FOR vnx IN var_numbers) row_j(vnx) -:= subtract*row(vnx); END;
            M(j) := row_j;
       END FOR j;
       var_order WITH := vn;          $ Note variable just processed
       var_numbers LESS := vn;        $ and exclude it from further processing.
    END FOR i;
  $ Next we work through the variables in the reverse
  $ order from which they were initially processed while calculating
  $ their values.  Note that at this point, the set var_numbers
  $ has become empty.
  (FOR i IN [n,n-1..1])
     row := M(i);
     vn  := var_order(i);
     soln(vn) := (row(n+1) -/ [soln(vnx) * row(vnx): vnx IN var_numbers])
                 /row(vn);
     var_numbers WITH := vn;
  END FOR;
  RETURN soln;                    $ Return the normal solution.
```

```
END PROC Gauss;
```

5.4.2 An Interactive Text-editing Routine

Our next example will serve to illustrate some of the internal workings of an interactive text editor (though actually the program to be given will support only a few of the features which a full-scale editor would provide, and even these are highly simplified). This editor has the following capabilities:

(a) A vector of strings representing a text file to be edited can be passed to it.

(b) The editor prompts its user for a command by printing '?', and waits for him to respond.

(c) The allowed responses are as follows:

(i) A response of the form '/ABCD..E/abc..e' makes ABCD..E a member of a collection of search strings that the editor maintains and indicates that some of the occurrences of ABCD..E in the text file are to be replaced by abc..e. Note that here ABCD..E and abc..e are intended to represent arbitrary strings which need not be of the same length; abc..e can even be null. Moreover, the 'delimiting character', which we have written '/', can be any character which does not appear in ABCD..E.

(ii) A response of the form '/ABCD..E' with just one occurrence of the initial 'delimiting character' indicates that ABCD..E is no longer to be searched for.

(iii) A response of the form '//' indicates that searching is to start again from the beginning of the text file. A response of the form '//done' indicates that editing is complete and triggers a return from the edit procedure.

(iv) A nullstring response searches forward in the text file for the next following occurrence of any search string ABCD..E. If any such occurrence is found, it is displayed on the user's terminal, with a line of 'underscore' characters placed immediately above it to mark its position. After this, another null response will trigger a search, but the response '/' will replace the string ABCD..E that has just been found by the corresponding string abc..e.

```
PROC edit(RW text);                    $ Text editor routine.
line_no := line_pos :=1;               $ Start at the first  character of
                                       $ the first  line of the text file.
replacement:=search_strings:={ };      $ Initially no search strings have
                                       $ been defined.
last_pos := OM;                        $ last_pos will be the last character
                                       $ position in a zone located by
searching;
                                       $ See the -search- procedure below.
                                       $ Initially, this is undefined.
```

```
first_chars := '';          $ first_chars is a string  consisting of
                            $ the first characters of all search strings.

LOOP DO
   IF (r := response( ))/='' THEN  $ Search forward from current position
      search(line_no, line_pos, last_pos, search_strings,
                                          first_chars, text);
      $ See the -search- procedure given below for an account of its
      $ parameters.

   ELSEIF #r = 1 THEN             $ Try to make replacement.

      IF last_pos = OM THEN       $ Successful search did not precede
                                  $ replacement.
         print('**NO SEARCH POSITION HAS BEEN ESTABLISHED **');
      ELSE                        $ Perform replacement.
            text(line)(line_pos..last_pos) :=
               replacement(text(line)(line_pos..last_pos));
      END IF;

   ELSE             $ The user's response was at least two characters
      c := r(1);    $ long.  Get first character of this response.

      IF NOT EXISTS i IN [2..#r] | c = r(i) THEN  $ Drop search string.

            replacement(strg := r(2..)) := OM;
            search_strings LESS := strg;
                  $ Recalculate the 'first-chars' string.
            first_chars := +/{x(1): x IN search_strings};

      ELSEIF #r = 2 THEN          $ '//'; hence restart search at top.

            line_no := line_pos := 1;
            last_pos := OM;       $ Invalidate search position.

      ELSE          $ A new replacement is being defined.

            replacements(strg := r(2..i-1)) := r(i+1..);
            search_strings WITH := strg;
                  $ Recalculate the string.
            first_chars := +/{x(1): x IN search_strings};
            last_pos := OM;        $ Invalidate any prior search.

      END IF NOT;

   END IF;

END LOOP;

END PROC edit;

PROC SEARCH (RW line_no, RW line_pos, RW last_pos, search_strings,
                                      first_chars, text);

   $ This procedure searches forward, starting at a given text line
```

```
$ and given character position, for the first position P at which
$ any member of the set -search_strings- of strings occurs. If such a
$ position is found, then -line_no- is set appropriately,
$ -line_pos- is set to p and -last_pos- is set to the index
$ of the last character matched.  If no such position is found,
$ then -last_pos- becomes OM while -first_pos- and -line_pos-
$ remain the same.

$ If -last_pos- is not OM, indicating that a successful search
$ has just taken place, then the search starts one character after
$ -line_pos-; this prevents repetitive searching.

search_string := text(line_no)(line_pos+1..);

(WHILE line_no <= #text)

   (WHILE search_string /= '')
         $ While a portion of the current line remains to be examined.

         IF break(search_string, first_chars) = OM THEN
            $ No significant character in this line, so go to next line.

            search_string := '';

         ELSE    $ See if one of the strings we are
                 $ looking for is found here.
                 TO BE CONTINUED
```

5.4.3 A Simplified Financial Record-keeping System

Next we will give SETL code representing some small part of the operations of a bank, albeit in simplified form. The system to be represented corresponds in a rough way to the 'Checking Plus' service offered by Citibank in New York City. Note, however, that the simple code shown below does not deal adequately with all the anomalies and error conditions that a full scale banking system would have to handle, nor does it support all the functions that are actually required. For example, the code we give does not provide any way for customer accounts to be opened or closed. A more ambitious commercial application showing how such matters can be treated is given in Section 5.4.3, but since the issues that enter into the design of a full-scale commercial system can grow to be quite complex, we prefer for the moment to evade many of them.

The simplified system which we consider is aware of a collection of customers, each of whom has an account. A customer's account consists of two parts, a balance representing funds available to him, and an overdraft debit representing the amount that he has drawn against the 'Checking Plus' feature of his account. This debit is limited for each account not to exceed a given -credit_limit-, established when the account is opened. The bank pays 5% per annum daily interest on positive balances in checking accounts, and charges 18% per annum daily interest on overdraft debits.

Like most commercial application programs, the code shown below maintains a 'data base', i.e. a collection of maps which collectively represent the situation with which the program must deal, and reads a 'transaction' file whose entries inform it of changes in this situation. Using these files it produces various output documents, for example, lists of checks deposited for transmission to other banks, monthly statements which are mailed to customers, etc.

The transactions supported by our simplified system are as follows:

TRANSACTION	CODE	EXPLANATION
deposit	(D)	Customer deposits either cash, a check drawn on another bank, or a check drawn on this bank.
withdrawal	(W)	A customer appears at a teller's booth and attempts to withdraw cash.
payment	(PA)	Customer transfers a stated sum from his available balance to reduce his overdraft debit.
presentation	(P)	Check is presented by another bank for payment.
clear	(C)	Another bank informs this bank that a check has cleared for payment.
return	(R)	A previously deposited check, sent to another bank for payment, is returned either as a bad check or for lack of available

		funds. (Checks written without sufficient funds cause their author's account to be debited $5.00).
end of day	(DAY)	End of banking day has arrived; daily interest is to be credited/ debited to all accounts.

On the last day of each month, an -end_of_day- transaction triggers the production of bank statements which are sent to each customer. On the last day of December, this statement includes an indication of interest charged and interest earned during the year.

Each transaction handled is represented by a single line (string) in the transaction file. This line always starts with a code letter identifying the transaction, and for the rest consists of various 'fields', separated by blanks. The fields expected for the various transactions supported are as follows:

D	customer_name	amount	bank_number (missing if cash deposit)	account_number
W	customer_number	amount	teller_terminal_number	
PA	customer_number	amount		
P	customer_number	amount	check_number	bank_number
C	check_number			
R	check_number	reason		
DAY				

The continuing data structures used to support our simplified banking system are as follows:

(1) cust_info — This map sends each customer_number into the record maintained for the corresponding customer.

The components of a customer record are:

balance_available	balance currently available
balance_deposited	balance showing checks deposited but not yet cleared
overdraft_debit	amount currently drawn against 'Checking Plus'
overdraft_limit	maximum overdraft allowed
transactions_this_month	list of all completed tansactions this month
interest_earned	total interest earned this year
interest_paid	total interest paid for overdrafts this year
name	customer name
social security number	customer social security number
address	customer address
telephone number	customer telephone number

(2) bank_info — This map send the numerical code of each bank from which checks will be accepted into the bank's address information.

(3) pending_checks — When check deposited are sent along to another bank for confirmation of payment, they are issued unique numerical identifiers. This maps sends

each such identifier into the transaction to which it corresponds.

Having now outlined all the transactions which our simplified banking system will support and listed the principal data structures which it uses, we are in position to give the code itself.

```
PROGRAM bank_checking;  $ simplified check-processing program
$ ******   DECLARATION OF GLOBAL VARIABLES, MACROS, AND CONSTANTS *******

VAR                     $ global variables

   Cust_info,           $ maps account number into customer record
   Bank_info,           $ maps bank number into bank address, etc.

   Pending_checks,      $ maps each suspended transaction numbers into
                        $ detailed transaction record

   This_banks_code,     $ code identifying this bank
   Check_counter,       $ counter identifying checks sent to other banks for
                        $ verification

   Message_list,        $ maps each bank identifier into a list of
                        $ messages to be sent to the bank.

   Bad_transactions,    $ accumulated list of bad transactions

   Last_day;            $ last day for which 'DAY' operation
                        $ was run

MACRO  customer_items;  $ The vector of items constituting a customer's
                        $ record. Note that all amounts are kept
                        $ as integer numbers of pennies.

[balance_available, balance_deposited, overdraft_debit,
      overdraft_limit, transactions_this_month, interest_earned,
                  interest_paid, name, sec_no, address, tel_no]

ENDM;

      CONST                 $ strings indicating transaction results
        CASH_DEP,           $ cash deposit
        CASH_WITHDRAWAL,    $ cash withdrawal
        PAYMENT,            $ payment of check
        OVERDRAW,           $ charge for overdrawn check
        NOFUNDS,            $ funds not available to pay check
        BAD_CHECK;          $ check drawn on nonexistent account

      CONST Transaction_codes = {D,W,PA,P,C,R,DAY};
        $ Constants designating transactions.

      CONST Involves_customer={D,W,PA};
        $ Transactions whose second parameter is a customer number.
```

```
      CONST Needs_updating = {D,W,PA,P,C,R};
        $ Transactions which modify customer record.

      CONST DIGITS = '0123456789'; $ the decimal digits

      CONST Annual_rate=6,       $ interest paid on checking balances
            Overdraft_rate=18; $ interest charged on overdrafts

        $ ****** MAIN PROGRAM OF BANKING SYSTEM ******

initialize_system; $ call initialization procedure to read in
                   $ all required global data structures.

      LOOP DO

        get(transaction);       $ read next transaction
        IF EOF THEN quit; END; $ all transactions processed
        process_transaction(transaction); $ otherwise process
                                          $ transaction
      END LOOP;

      finalize_system; $ write state of system to output file

      print; print; print('END OF TRANSACTION PROCESSING');

PROC process_transaction(t);  $ The principal transaction-processing
                              $ procedure.

IF (dec := decode_transaction(t)) = OM THEN RETURN; END;
         $ Since transaction is bad.

[code, number, amount, p4, p5] := dec;     $ Get fields of transaction.

IF code IN Involves_customer THEN $ Obtain fields of customer record.

    customer_items := Cust_info(number);
    $ Make balance_available, balance_deposited, overdraft_debit,
    $ overdraft_limit, etc. available.

END IF;

  CASE code OF

(D):  $ deposit

    IF p4 = OM THEN $ deposit is cash: accept it immediately

      balance_available +:= amount;
      balance_deposited +:= amount;
      transactions_this_month WITH:= post(CASH_DEP,amount);

    ELSEIF p4 = This_banks_code THEN $ check is drawn on this bank

      $ We handle a check drawn on this bank as a
```

```
      $ combination of a 'P' transaction with the transaction (either
      $ 'C' or 'R') that responds to this 'P' transaction.  For this,
      $ it is convenient to allow this procedure to call itself
      $ recursively.

     balance_deposited +:= amount;
     pending_checks('0') := t;

      process_transaction('P ' + p5 + ' ' + dollar(amount) + '0'
                                                  +This_banks_code);
      result := Message_list(This_banks_code)(1); $ Get result and
      Message_list(This_banks_code) := [ ];       $ clear message list

      process_transaction(result); $ proccess the resulting 'C' or 'R'

    ELSE $ The check is drawn on another bank.  Note, but do not
         $ credit, the deposit.

      balance_deposited +:= amount;
      identifier := STR (Check_counter +:= 1);
      Pending_checks(identifier) := t;  $ Save transaction for
                                        $ later completion.

      Message_list(p4) WITH:= $ send nofification to bank on which
                              $ the check is drawn
'P ' + p5 + ' ' + dollar(amount) + ' ' + identifier +
                                       ' '    +This_banks_code;

    END IF;

(W): $ Withdrawal

    IF ok_withdraw(amount,balance_available, overdraft_debit,
                           balance_deposited, overdraft_limit) THEN
       send_teller(p4, 'PAYMENT APPROVED');
       transactions_this_month  WITH:= post(CASH_WITHDRAWAL,amount);
    ELSE
       send_teller(p4,NOFUNDS);
    END IF;

(PA): $ payment of portion of overdraft debit

    will_pay := amount MIN balance_available MIN overdraft_debit;
    balance_available -:= will_pay;
    balance_deposited -:= will_pay;
    overdraft_debit -:= will_pay;
    transactions_this_month WITH:= post(PAYMENT,will_pay);

(P):  $ presentation (for approval) of check by other bank

    IF(c_info := Cust_info(number))=OM THEN $ check is bad
        Message_list(p5) WITH := 'R '+ p4 + ' '+ BAD_CHECK;
           RETURN;              $ abort transaction
    END IF;
```

```
    customer_items := c_info;  $ make fields of customer info available

    IF ok_withdraw(amount,balance_available,overdraft_debit,
                                balance_deposited,overdraft_limit) THEN

        Message_list(p5) WITH:= 'C '+ p4; $ confirm clearance
        transactions_this_month WITH:= post(PAYMENT,amount);

    ELSE

        Message_list(p5) WITH := 'R '+ p4 + ' '+ NOFUNDS;

       $ In this case the customer is charged a $5.00 fee,
       $ or whatever smaller amount remains in his account

       charge := 500 MIN (balance_available+overdraft_limit-
                                              overdraft_debit) MAX 0;

       ASSERT ok_withdraw(charge,balance_available,overdraft_debit,
                                         balance_deposited,overdraft_limit);

       transactions_this_month WITH:= post(OVERDRAWN,charge);

    END IF;

(C):  $ pending check clears

      ASSERT(dec := decode_transaction(Pending_checks(number)))/=OM;

       $ We can make this assertion because the system
       $ represented here does not allow customer accounts to be
       $ closed.  However, this assertion would continue to hold true even
       $ in a more realistic system, since in such a system we would not
       $ close an account until all its outstanding deposit transactions
       $ have been completed.

       Pending_checks(number) := OM; $ drop from pending list
       [-,-,amount] := dec;
       customer_items := Cust_info(number);
       balance_available +:= amount; $ credit to available balance
       transactions_this_month WITH:= post(CHECK_DEP,amount);

(R):   $ pending check fails to clear

       reason := p4; $ in this, case the p4 field contains the reason
                     $ for refusal of the check transmitted for approval

       ASSERT (dec := decode_transaction(Pending_checks(number)))/=OM;
       $ see comment following case(C)

       Pending_checks(number) := OM; $ drop from pending list
       [-,-,amount] := dec;
       customer_items := Cust_info(number);
       balance_deposited -:= amount; $ Debit the estimated total
                                     $ of deposits.
```

```
        transactions_this_month WITH:= post(reason,amount);

(DAY): $ End of banking day: take end-of-day, and if necessary
       $ end-of-month, actions.

       end_of_day; $ take end of day actions

       IF day(DATE) = '1' THEN
         end_of_month;
       END IF;

       ELSE $ have some system error.  Take end_of day action,
            $ save system, and note error.

       print('SYSTEM ERROR *** ILLEGAL TRANSACTION:', t);

       end_of_day;
       finalize_system;

       STOP;

END CASE;

IF code IN Needs_updating THEN    $ customer information must be updated
    Cust_info(number) := customer_items;
END IF;

    END PROC process_transaction;

    PROC ok_withdraw(amount,RW bal_avail, RW over_debit, RW bal_deposit,
                                                    over_limit);

    $ This auxiliary procedure checks to see if the stated -amount-
    $ can be withdrawn from an account, by increasing
    $ the overdraft debit if necessary.  If so, the balance
    $ available, amount provisionally on deposit, and the
    $ overdraft debit are appropriately adjusted,
    $ and TRUE is returned; otherwise FALSE is returned.

    IF amount > (avail + over_limit - over_debit)  THEN $ no good

       RETURN FALSE;

    END IF;

   bal_avail -:= (amt_frm_bal := amount MIN bal_avail);
   bal_deposit -:= amt_frm_bal;  $ decrement amount provisionally on deposit
   over_debit +:= amount - amt_from_bal;

   RETURN TRUE;

   END PROC ok_withdraw;

   PROC post(trans_type,amount);
```

```
$ This auxiliary routine converts transactions into strings consisting
$ of an amount, a coded indicator of the transaction type, and a
$ date; the result is suitable for printing in a customer's
$ end-of-month statement.

RETURN DATE + ' ' + trans_type + ' ' + dollar(amount);

END PROC post;

PROC decode_transaction(t); $ decodes string form of transaction

$ This procedure reads the string form of a transaction and
$ decodes it into the various blank-separated fields of which it
$ consists.  It verifies that each field has the expected type.
$ If any field is found to be bad, or if any field is missing, then
$ the transaction is posted to a 'rejected transactions' list, and
$ this procedure returns OM.  Otherwise, a tuple consisting of the
$ converted fields is returned.

CONST Check_strings = $ Map from transaction type to
                      $ pattern of fields expected for transaction.
                      $ See procedure -field_check-, below,
                      $ for an explanation of the codes appearing here.

  {[D,XCABX], [W,XCAX], [PA,XCA], [P,XXXXX],
                  [C,XX], [R,XXX], [DAY,X]};

  decoded_trans := [ ]; $ tuple for decoded form of transaction
  nfield := 0;          $ counter for field number

  check_string := 'T';  $ check character for first field is 'T'

  (WHILE t/= '' AND (nfield +:= 1)<6)

    IF SPAN(t,' ')/= OM THEN CONT; END; $ span off blanks

    IF (field := field_check(BREAK(t,' '),
                                Check_strings(nfield))=OM THEN

      Bad_transactions WITH:= t;
      RETURN OM;

    END IF;

  $ If the first field has just been decoded, use it to determine
  $ what further checks are necesary.

    IF nfield =1 THEN check_string := Check_strings(field); END;

    decoded_trans WITH:= field; $ otherwise store field

  END WHILE;

  $ Check that all required fields, and no others, are present.
```

```
IF #decoded_trans=#check_string
  OR decoded_trans(1) = D AND #decoded_trans=3 THEN
    RETURN decoded_trans;
END;

Bad_transactions WITH:= t;  $ Otherwise missing or superfluous
                            $ fields.
RETURN OM;

END PROC decode_transaction;


PROC field_check(field, test_char); $ auxiliary test/convert
                                   $ procedure

$ This procedure checks the -field- passed to it for conformity
$ with the expected field type, which is descibed by its
$ -test_char- argument.

$ The allowed test_char characters, and their significance,
$ are as follows:

$ 'T': must be transaction code
$ 'X': no test required
$ 'C': must be customer account number
$ 'A': must be dollar amount
$ 'B': must be identifier of correspondent bank

$ If the test fails, then OM is returned; if the test succeeds,
$ and the field type is 'A', then the field is converted from
$ standard DDDD.CC 'dollars and cents' form to an integer
$ number of cents,

    CASE test_char OF

  ('T'): RETURN IF field IN Transaction_codes THEN field
                                              ELSE OM END;

  ('X'): RETURN field;

  ('C'): RETURN IF Cust_info(field) = OM THEN OM ELSE field END;

  ('A'): dollars := SPAN(field,Digits)?'';

         IF MATCH(field,'.')=OM THEN RETURN OM; END;

         cents := SPAN(field,Digits)?'';
         IF #cents/=2 OR field/='' THEN RETURN OM; END;

         RETURN VAL(dollars + cents);

 ('B'): RETURN IF Bank_info(field)=OM THEN OM ELSE field END;

       ELSE
          RETURN OM;
```

```
END CASE;

END PROC field_check;

PROC initialize_system; $ system initialization code

$ First we acquire the name of the input file
$ for this run of the banking system, which is supplied as a
$ 'control-card' parameter; see Section 8.5.

input_file := getspp('OLD=OLD.DAT/OLD.DAT');

$ Next we read the code for this bank, the pending transaction
$ counter, the master customer file,
$ the bank address file, and the last previous processing date,
$ from the specified input information file.

OPEN(input_file,'CODED'); $ Open the input file for reading.
                          $ (See Section 8.1).

reada(input_file, This_banks_code,Check_counter, Cust_info,
                                                 Bank_info,Last_day);

CLOSE(input_file); $ now finished with input file; release it
                   $ (See Section 8.1).

$ Next various subsidiary initializations are performed.

Pending_checks := { };   $ pending check mapping is empty
Bad_transactions := [ ]; $ list of bad transactions is empty
Message_list := {[bank,[ ]] : x = Bank_info(bank)};
                         $ start an empty message file for each
                         $ correspondent bank

END PROC initialize_system;

PROC finalize_system; $ end-of-run 'dump' procedure

$ First we acquire the name of the output file for this run of
$ the banking system, which is supplied as a 'control card'
$ parameter; see Section 8.5.

output_file := getspp('NEW=NEW.DAT/NEW.DAT');

OPEN(output_file,'CODED-OUT'); $ open the output file for writing.
                               $ (See Section 8.1).

$ Next we write the code for this bank, the pending transaction
$ counter, the master customer file, and the bank file to the
$ specified output file

printa(output_file, This_banks_code, Check_counter, Cust_info,
                                                 Bank_info,DATE);
```

```
CLOSE(output_file);  $ now finished with output file; release it
                     $ (See Section 8.1).

END PROC finalize_system;

PROC send_teller(terminal_no,msg);

$ In an actual system, this procedure would send the message
$ -msg- to the teller terminal identified by -terminal_no-. Since
$ it is not easy to use SETL to send messages to more than
$ one terminal, we simplify this procedure drastically, and simply
$ print -msg-, with an indication of the number of the terminal
$ to which msg should actually be sent.

print(msg, 'has been sent to terminal', terminal_no);

END PROC send_teller;

PROC end_of_day; $ end of day procedure

$ This procedure is called at the end of each banking day
$ In practice, it would write out a collection of files,
$ including the following:

$ (a) for each bank with which this bank does business,
$     a file of messages, each representing either a

    $ (i) confirmation that a check transmitted for approval
    $     was actually approved;

    $ (ii) rejection of a check, with an indication of the reason
    $      for rejection;

    $ (iii) request for approval of a check,

$ (b) a list of bad transactions, for visual inspection and
$     possible re-entry.

$ We begin by crediting interest payments and making
$ interest charges for all customers.

$ First check to ensure that interest has not already been
$ credited today.

IF DATE /= Last_day THEN

   (FOR c_info = Cust_info(number))

          interest_earned +:=
          (earned := (balance_available*Annual_rate) DIV 36500);
          balance_available +:= earned;

          $ Next, make charges on the customer's overdraft debit
          interest_paid +:=
             (owed := (Overdraft_debit*Overdraft_rate) DIV 36500);
```

```
        $ Draw this interest out of the account if possible.
        $ If not enough remains, interest will be charged as an
        $ overdraft, even though this causes the actual overdraft to
        $ exceed its stated limit.

           IF NOT ok_withdraw(owed,balance_available,overdraft_debit,
                       balance_deposited,overdraft_limit) THEN

                $ run an 'excess overdraft'
                overdraft_debit +:= owed - balance_available;
                balance_deposited -:= balance_available;
                balance_available := 0;
           END IF;
           Cust_info(number) := c_info;

     END FOR;

END IF DATE;

$ Write a file of messages for each bank with which this bank does
$ business.

(FOR bank_inf = Bank_info(code) | code /= This_banks_code)

  write_message_file(bank_inf, Message_list(code));
  Message_list(code) := [ ];  $ clear the message list to avoid
                              $ resending.

END FOR;

$ Write out the file of bad transactions.
write_bad_transactions(Bad_transactions);
Bad_transactions := [ ];  $ clear the list of bad transactions

END PROC end_of_day;

PROC write_message_file(bank_inf, mess_list);

$ In a realistic system, this procedure might write
$ a list of messages to a magnetic tape which was then
$ sent by air-express or special courier to one of the banks with
$ which this bank does business.  However, in our simplified
$ system, we simply print out -bank_inf- as a header,
$ and follow it by the individual messages of mess_list.

print; print; print(bank_inf); print; print;

(FORALL m IN mess_list) print(m); END;

END PROC write_message_file;

PROC write_bad_transactions(list);

$ In a realistic system, this procedure might write its list of
```

```
      $ transactions to an on-line disk file, which would then be
      $ scrutinized and manually edited, reference being made if
      $ necesary to the original handwritten or typed document which
      $ first ordered the transaction.  However, in our simplified
      $ system, we simply print out the list of bad transactions.

      print; print; print('BAD TRANSACTION LIST'); print; print;

      (FORALL m IN list) print(m); END;

      END PROC write_bad_transactions;

      PROC end_of_month; $ end-of-month procedure

      $ This procedure, called on the last day of each
      $ month, prepares a monthly statement for each customer.
      $ If the month is January, a statement of total
      $ interest charged/earned appears on the statement,
      $ and the accrued interest fields in the customer record are
      $ cleared.

      IF DATE = Last_day THEN RETURN; END; $ since statements have already
                                           $ been prepared.
      is_January := (month(DATE) = '1'); $ test for January

      (FOR customer_items = Cust_info(cust_number))

          print; print(name,sec_no); print(address); print(DATE); print;

          (FOR trans IN transactions_this_month) print(trans);END;

          transactions_this_month := [ ];

          IF is_January THEN
            print;
            print('SAVE THIS STATEMENT-IT CONTAINS VALUABLE TAX'
                                        'INFORMATION');
            print;
            print('Interest earned:', interest_earned);
            print('Interest paid:', interest_paid);
        END IF;

    END FOR;

    END PROC end_of_month;

END PROGRAM bank_checking;
```

5.5 Exercises

Ex. 1 Write a program that will read a real number x and print the number of decimal positions of x which lie to the left of the decimal point.

Ex. 2 Write an expression which will take any SETL tuple t and generate a map f which indicates how many components of t are of type 'ATOM', 'BOOLEAN', 'INTEGER', 'REAL', etc., and how many components of t are OM.

Ex. 3 Which of the following operations will cause an error:

```
(a)  2.2 (1.1 + 1.10-2.200)
(b)  -2.2*-2.2**2.2
(c)  (-2.2)**2.2
(d)  FLOAT(-2)*2
(e)  (-2.2 MAX 2.2)**-2.2
(f)  (-2.2 MIN 2.2)**2.2
(g)  SQRT(-2 MAX 2)
```

Ex. 4 Test the following boolean expressions to see if they yield TRUE or FALSE:

```
(a)  1.0=2.0-1.0
(b)  2.0=SQRT(4.0)
(c)  SIN(ASIN(0.5))=0.5
(d)  SIN(0.5)*SIN(0.5)+COS(0.5)*COS(0.5)=1.0
```

Determine the size of the difference between the left and the right hand side of each equality which yields the value FALSE.

Ex. 5 Which of the following statements are true for all values of the variable x?

```
(a)  ABS(FLOAT(x))=FLOAT(ABS(x))
(b)  FIX(FLOAT(x))=FLOAT(FIX(x))
(c)  FLOOR(x) < FIX(x)
(d)  CEIL(x) >= FIX(x)
(e)  EXP(LOG(x)) = x
(f)  LOG(EXP(x)) = x
```

Ex. 6 For what positive values of x is COS(x) closest to 0.0? What is the value of ASIN(1.0)? Check your answers by computer evaluation.

Ex. 7 How small is the sum SIN(x) + SIN(x + 3.1415928)? (Evaluate it at the points x=-3.1415928, 0.0, 3.1415928, etc.) Can you find a constant c such that SIN(x) + SIN(x+c) is smaller than SIN(x) + SIN(x+3.1415928) for several values of x?

Ex. 8 Square the quantity x:=2.0/SQRT(4.0) repeatedly to see how its higher powers behave. How many squarings are required to calculate x**1024?

Ex. 9 Write the values for which x, y and z will have after each of the following sequences is executed.

```
(a)  x:='abc'; y := SPAN(x,'ABC');
(b)  x:='abc'; y := ANY(x,'ABC');
(c)  x:='abc'; y := SPAN(x,'ab'); z := RANY(y,'ab');
(d)  x:='abc'; y := BREAK(x,'ABC');
(e)  x:='abc'; y := BREAK(x, 'abc');
(f)  x:='abc'; y := RBREAK(x,'ABCabc');
(g)  x:='abc'; y := LEN(x,4);
(h)  x:='abc'; y := NOTANY(x,'ABC');
(i)  x:='abc'; y := RNOTANY(x,'ABC');
```

Ex. 10 Write a program which will read a string s and will

(a) delete all sequences of blank spaces immediately preceding a punctuation mark,

(b) insert a blank space immediately after each punctuation mark that is not followed by either a blank or a numeric character.

Ex. 11 Write a program which prints a set s of words in an alphabetized, neatly formatted arrangement; the words printed should be lined up in rows and columns. As many columns as possible should be used, but at least two blank spaces must separate any two words printed on the same line.

Ex. 12 Modify the lexical scanner procedure of Section 5.2.1.1 so that it returns a pair [toks_and_types,val_map], where toks_and_types is a tuple of pairs [tok,tok_typ], each -tok- being a token appearing in the source text scanned, and -tok_typ- is the type (i.e. 'INTEGER', 'REAL', 'IDENTIFIER', or 'SPECIAL') of -tok-. The quantity -val_map- should be a map sending the string form of each integer and real number appearing in the sequence of tokens to its value.

Ex. 13 As written, the lexical scanner procedure of Section 5.2.1.1 always treats the underbar character as a special character and does not allow real numbers like '.3' which begin with a period. Modify this procedure so that it allows underbars within identifiers (but not as the first character of identifiers) and allows real numbers to start with the '.' character.

Ex. 14 Modify the concordance program shown in Section 5.2.1.2 so that

(a) all words less than three characters long are omittted from the concordance;

(b) the program begins by reading a list of 'insignificant' words which occur on a sequence of lines terminated by a line containing the string '*****'. It then omits them from the concordance. (Multiple insignificant words can also occur, separated by blanks on a single line).

Ex. 15 Modify the concordance program shown in Section 5.2.1.2 so that it begins (cf. Exercise 14) by reading a blank-separated list of words, and reports only on the occurrences of words belonging to this list.

Ex. 16 Modify the concordance program shown in Section 5.2.1.2 so that it reports only on 'infrequent' words, i.e. words that occur no more than twice. Words belonging to a specified set s of words should be ignored even if they are infrequent. Programs of this kind can be used to locate

'suspicious' identifiers in other programs, i.e. identifiers which may have been misspelled or simply forgotten during program composition.

Ex. 17 The simplified text editor shown in Section 5.2.1.3 does not protect its user against any of the errors that are likely to occur during a lengthy edit session. Add code which will alleviate this deficiency by implementing the following additional features:

(a) Demand that '//', rather than any arbitrary string of two identical characters, be used to restart editing from the first line of the file F being edited, and that '/', rather than any arbitrary one character string, be used to trigger a replacement.

(b) Allow an additional command 'x', which should produce a formatted display of all search strings, with their replacement strings.

(c) Allow an additional command 'f', which should undo the last correction made. Your system should allow up to five successive changes to be undone using the 'f' command.

(d) Allow the command '\' to trigger a search backward through the file, i.e. a search from the current character position through earlier positions and lines.

Ex. 18 Browse through the user's manual of some text editor of medium complexity to become familiar with the various features it provides. Select an interesting one of these features, and modify the text editor code shown in Section 5.2.1.3 so that it implements the feature which you have selected.

Ex. 19 The function SIN(x) is the sum of the infinite power series whose n-th term is ((-1)**n)*(x**(2*n+1))/(2*n+1)! (n ranges upward from 0).

(a) Let S5(x) and S10(x) denote the first five and first ten terms of this series respectively. Calculate and print the difference S5(x)-SIN(x) and S10(x)-SIN(x) for each value of x from 0.0 to 3.14159 by steps of 0.1. What maximum deviation between S5(x) and SIN(x) do you find? Can you find a constant b such that addition of b to S5(x) reduces this maximum deviation?

(b) Repeat part (a) for COS(x). This is the sum ofthe infinite series whose n-th term is ((-1)**n)*(x**(2*n))/(2*n)! (again, n ranges upward from 0).

Ex. 20 Modify the character-string search procedure shown in Section 5.2.1.3 so that it can locate strings which run over from one line to the next. How should the editor program of Section 5.4.2 be modified to allow easy editing of strings of this sort?

Ex. 21 Certain types of forests are subject to infestation by budworms. The following rules can be used to model the results of such an infestation. We suppose for simplicity that the forest consists of an n by m rectangular array of trees. In a given year, any tree will be either healthy, infested, or leafless, having been infested the year before. A tree infested one year will be leafless the next year; a tree leafless one year will be healthy the next year. A tree healthy one year will be healthy the next year unless

its neighbor to the North, South, East, or West is also infested, in which case it will also become infested the next year.

Write a program which will simulate the progress of a budworm infestation obeying these rules. Track the progress of an infestation which starts with just one infested tree, and the progress of an infestation that starts with a row of three infested trees. Your program should print out a diagram of the forest in each of a sequence of years, together with a count of the number of infested, leafless, and healthy trees.

Ex. 22 Write a procedure which can be used to print a coarse 'graph' for any real-valued function f of a real variable x. This should be written as a procedure with real parameters lo, hi (the lower and upper limit of the values of x for which f(x) will be graphed), lo_range, hi_range (the lower and upper limits of the range of f that will be graphed), and an integer parameter n (the number of lines on the printed output listing that the graph should occupy). Your procedure should call a subprocedure, 'f_to_graph' to obtain the values of the function to be graphed. Vertical and horizontal axes should be printed with the vertical axis at the extreme left of the output listing. These axes should carry suitable markings to indicate the scale. The x-axis should run horizontally.

How would you change this procedure if the x-axis is to run vertically down the length of the output listing?

Ex. 23 Write a procedure which can be used to print a graph showing the values of several functions f(x). The main input to this procedure should be a sequence of tuple t of real numbers all having the same length. Each of these tuples will reperesent a sequence of values of one function f(x). Auxiliary inputs will be two real numbers, -lo- and -hi-, defining the minimum and maximum values of the domain over which the dependent variable x has been evaluated to produce the tuple t, a character string whose jth character will be used to print points belonging to the graph of the jth function, and an integer n indicating the number of lines of the output listing which the graph is to occupy. Your procedure should be written to accept various numbers of tuples t. The scale of the graph should be adjusted to reflect the largest and the smallest values appearing in any of the tuples t. Axes should be printed with scales marked on both the x and y axis. If the tuples t are too long to be displayed with the x axis running horizontally, the graph should be turned 90 degrees so that the x axis runs vertically down the listing.

Ex. 24 Write a procedure P which can be used to generate a variety of commercial reports in graphical form. The inputs to P should be two tuples, t1 and t2, of sales or production figures; t1 representing the 'current year' and t2 the 'prior year'. The third parameter of P should be a two character string defining the bar chart desired, encoded in the following way:

'm' - monthly figures desired
'c' - cumulative monthly figures desired
'd' - difference between current and previous year desired
'p' - percentage difference between current and previous year desired.

The 'd' chart should be organized as a series of adjacent pairs of bars

showing figures of the current year and the previous year. Axes should be printed with the vertical axis using an appropriate scale and the horizontal axis carrying the names of the months. The 'p' chart requires only a single bar for each month. What other useful features can you design and implement for a program of this kind?

Ex. 25 Write a procedure which prints 'bar charts' or 'histograms'. The inputs of this procedure should be a tuple t of real numbers and an integer n indicating the number of lines on your listing that the chart is to occupy. A set of bars representing the components of t in graphic form should be printed. The scale of the bars should be adjusted to reflect the largest component and the smallest component of t, and the thickness of the bars should be adjusted to the length of t and the number of columns available on the output listing. Axes should be printed with the vertical axis being scaled. If t is too long for the required number of bars to fit horizontally, the chart should be turned 90 degrees so that the bars of the chart are horizontal.

Ex. 26 Generalizing the procedure of Exercise 25, write a procedure which prints bar charts with bars which are divided into different 'zones' representing different sets of quantities. The main input to this procedure should be a sequence of tuples t of real numbers all having the same length. (But think of a good way to handle the case in which not all tuples have the same length!) The auxiliary inputs to the routine are a character string whose jth character will be used to print the jth zone of each bar and an integer n indicating the number of lines that the chart is to occupy on your listing. The procedure should be written to allow various number of parameters t. If the tuples t are too long for the required number of bars to fit horizontally, the chart should be turned 90 degrees so that the bars are horizontal.

Ex. 27 Write a procedure which can be used to print a graph of the 'level curves' or 'contours' for a real-valued function of two

TO BE CONTINUED

Ex. 28 Write a translation program which translates French to English word-by-word. (Warning: such a program will produce extremely mediocre translations). The program should read a file of lines containing successive blank-separated pairs of French words and their English translations, and then read a French passage to be translated and print out its English translation.

Ex. 29 Modify the word-by-word translation program described in Exercise 28 so that it becomes interactive, and so that it is prepared for the fact that certain French words might have several possible translations into English. When such words are encountered during translation, a numbered menu of all of them should be displayed, and the user should then have the ability to continue by selecting one of these possible translations.

Ex. 30 PERT charts are used by project administrators to track progress and monitor critical activities in large projects. To set up such a chart, one first reads in a set s of pairs [activity1, activity2] defining the collection of all activities that must finish before any given activity2 can start. One also reads a map T sending each activity to its expected

duration. Then one calculates the earliest time that each activity A can finish, and for each such A, the set of all activities whose completion is critical to completing A by this time. Then one can print a list of all activities in order of their completion times. Then, working back from the final activity which marks the completion of the whole project, one can calculate the set of all critical activities, that is, all activities which must be completed on time if completion of the whole project is not to be delayed. One can also calculate and print the degree of 'slack' available for each activity, i.e. the amount that its completion could be delayed without slowing completion of the whole project.

Develop a program that calculates this information and prints it out in a set of attractively formatted tables.

Ex. 31 (Continuation of Exercise 30) Once started, large projects often begin to 'slip' because some of their critical activities are not completed on time. Modify the PERT program of Exercise 30 to allow it to read a list of activities which have already been started, together with their expected completion times, a new list of critical activities, and a revised table of 'slack for all (started and unstarted) activities. Can you design and implement any additioanl features which would invoke this PERT program more a useful planning tool, especially if it is to be used interactively?

Ex. 32 Write a program which will generate the integers from 4000 through 4100 and print them out with appropriate check characters (See Exercise XXX) appended. Write another program which will read in items to which check characters have been appended and reject items in which errors are detected. For the integers from 4000 through 4100 with check characters appended, see how many will yield undetected errors if one digit is mistyped or if two digits are transposed.

Ex. 33 Add code to the banking system shown in Section 5.4.3 so that it insists that a customer pays at least five percent of his overdraft debit by the end of each month and adds a warning notice to his monthly statement if this is not done.

Ex. 34 Add code to the banking system shown in Section 5.4.3 so that it can handle a 'report' transaction (R), which generates a report of the total number of transactions handled and the total dollar volume of transactions handled, by category of transaction for each hour of the current day, up to the current hour.

Ex. 35 A confidence man trying to pass forged checks drawn on an account which is not his own may try to go from one teller to another (in one or more branches of a bank) cashing checks repeatedly at teller windows. Add code to the banking system shown in Section 5.4.3 which will make this more difficult to accomplish. Your code should keep track of all withdrawals made from a given account within two days before the current day. If these withdrawals add up to more than 20% of the total amount that can be drawn from the account, the teller entering a withdrawal should be alerted by receiving a statement of the total number of checks withdrawn on each of these days, and of the total amount withdrawn on each of these two days. Withdrawals for which this warning is issued should be held until the teller sends in a go (G) transaction and should be dropped if the teller sends in a kill (K) transaction instead.

Ex. 36 Write a print_monthly_statement procedure invoked near the end of the -process_transaction- procedure shown in Section 5.4.3. Your procedure should print this statement in an appealing format, appending end of year information and warning messages as required.

Ex. 37 Write the print_interbank_balances procedure invoked near the end of the -process_transaction- procedure shown in Section 5.4.3. This should list balances due to and from other banks in two ways, the first sorted in decreasing order of net amount due to/from other banks, the second sorted alphabetically by the name of the bank. The total change in sums due to/from other banks should also be reported.

Ex. 38 A meteorological station measures the temperature every hour, producing records arranged as a sequence of tuples t, each t having length 24 and representing a day's temperature measurements (the first being taken at midnight). Write a program which will read this data and print out a record of the highs, lows, and mean temperature for the entire day, and also the highs, lows and mean temperature for the 'daylight' hours (7 AM through 6PM).

Ex. 39 The bureau of crime statistics receives annual reports from all cities and incorporated towns, showing the number of major felonies recorded for the year. It then calculates the total number of cities and towns reporting felonies in the ranges 1-100, 101-500, 501-1000, 1001-2000, and more than 2000. Assume that the file of data being read is a set of cards, each of which contains the name of a town and the number of reported felonies, separated by a blank, write a program for preparing and printing this report.

Ex. 40 When commands need to be entered interactively at a terminal, it is convenient to allow the shortest unambiguous prefix of any command to serve as an abbreviation for the command. Write a procedure which makes this possible. (Hint: alphabetize the set of allowed commands and locate prefixes by a fast search in this alphabetized list).

Ex. 41 Large sets of alphabetic strings which need to be stored can be represented in compressed form by arranging them in alphabetical order. Then all the strings beginning with a particular character, say 'a', can be preceded by the string '1a', and the initial letter 'a' dropped from all of them. Similarly, if the group of strings beginning with the letter 'a' contains more than two successive strings whose second character is 'b', then the whole group of such strings can be prefixed by the string '2b', and theinitial letters 'ab' dropped from all of them. This transformation can be applied to as many initial characters as are appropriate.

Write a procedure which takes a set s of strings, alphabetizes it and compresses it using this technique. Write another procedure which takes a set s of strings represented in this form and prints s in its original alphabetized form.

Ex. 42 Generate about a hundred random pairs of tuples, t1 and t2, of the same length, all of whose components are real numbers. Then count the number of those t's which satisfy the following inequality:

```
(+/[x*x: x IN t1])*(+/[x*x: x IN t2])
```

```
< ABS(+/[t1(i)*t2(i): x IN [1..#t1]]).
```

(Be careful not to be fooled by small errors in the computation, i.e. a pair of tuples that barely satisfies or fails to satisfy the preceding equality should be considered indeterminate and ignored). What percentage of the tuples tested satisfy this inequality? What do you deduce from this?

Ex. 43 Run the following programs and see what results they produce

(a) `x:=2.0; (FOR n IN [1..100]) x:=x*x; print(n,x); END;`

(b) `x:=0.5; (For n IN [1..100]) x:=x*x; print(n,x); END;`

Ex. 44 Build and print out the following sets, letting x vary over 10 real numbers chosen at random from the range 1.0 to 10.0:

(a) The set of x for which `x**0` or `x**0.0` is different from 1.0.

(b) The set of x for which `x**0` or `x**0.0` is different from 1.

(c) The set of all differences `sqrt(x)-x**0.5`.

(d) The set of all differences `x**0.5 - x**(1.0/2.0)`.

(e) The set of all differences `x*x-x**2.0`, and the

set of all differences `x**2-x**2.0`.

(f) The set of all differences `x-(x**3)**(1.0/3.0)`.

(g) The set of all differences `x*x/x-x`.

(h) The set of all x such that `SIN(x)**2+cos(x)**2=1.0`.

Ex. 44 Write a short program which would work perfectly if perfectly accurate real arithmetic were performed but which fails catastrophically because of small inaccuracies in the computer representation of reals.

$

CHAPTER 6

ADDITIONAL CONTROL-LIKE FEATURES OF THE SETL LANGUAGE

In the present, relatively short, chapter we round out our account of the control structures of SETL by describing certain useful facilities not covered in earlier chapters.

Chapter Table of Contents

6.1 Refinements

By now you will be familiar with the general process of program development. Starting from the description of a problem, one sketches out a general approach, breaks the problem into simpler subproblems, and then refines an initial program sketch until a full program, complete in all its details, emerges. This process of stepwise refinement is central to programming: breaking down the initial problem into more manageable pieces is the only way we have of coping with really complex tasks. Without some machinery to help us in this subdivision process, it would be impossible (not just difficult) to write large programs. The main tool used to decompose a problem into relatively independent components is that presented in Chapter IV, namely the use of functions and procedures, which communicate with each other by passing parameters and returning values. However, in some cases there is no need to cling to this parameter-passing discipline: the problem breaks down into a simple sequence of actions which can as well be made global. For use in such cases, SETL provides a different syntactic

mechanism: the REFINEMENT, which allows the user to name groups of actions in the procedure in which they are used and to invoke them by name .

A refinement is a sequence of SETL statements, preceeded by a name and two colons, as in the following example:

```
converge::
        x := 0 ;
        y := 1 ;
        z := fun(x,y,100) ;
```

A procedure which uses refinements names the refinements it uses in the order in which they are used. A given refinement can only be invoked once. If a given sequence of statements is to be used more than once, it must be made into a procedure. The following example shows the use of refinements.

```
program roots ;          $ Calculate and print roots of quadratic equation.
get_coefs ;              $ Step 1: get coefficients of the equation.
eval_discr ;             $ Step 2: evaluate the discriminant of the equation.
eval_roots ;             $ Step 3: evaluate the roots of the equation.
print_roots ;            $ Step 4: print the roots.
                         $ Next follows the four refinements just invoked.

get_coefs:: read(a,b,c) ;
            IF a = 0 THEN print('Degenerate case') ; STOP ;
            END IF ;

eval_discr:: d := b ** 2 - 4.0 * a * c ;
             IF d < 0 THEN
                print('Complex roots') ; stop ;
             ELSE
                discr := sqrt(d) ;
             END IF;

eval_roots:: x1 := (-b + discr)/(2.0 * a) ;
             x2 := (-b - discr)/(2.0 * a) ;

print_roots:: print('First  root: ', x1) ;
              print('Second root: ', x2) ;

stop ;
```

Execution of a program with refinements proceeds as if the body of the refinement (the statements that follow the double colon) had been inserted at the point at which the refinement is named. Note that the refinements have no parameters and need contain no RETURN statements. They are in the same scope as the procedure, module or main program in which they appear; thus they have access to all the identifiers that are visible in this procedure, module or main program. In the example above, the variables x1, x2 and discr are used in several refinements and could also be used in the main program which invokes these refinements.

Refinements allow us to write a procedure or program with a 'table of contents' as it were. To someone who is only interested in the rough structure of an algorithm, reading only the names of the refinements and the comments attatched to these names may be sufficient to gain a quick understanding of its workings. This requires that the names chosen for the refinements be meaningful and reflect their purpose. Here, as elsewhere, a judicious choice of names will add significantly to the understandability of a program.

6.3 The CONST Declaration

It is often convenient to use a symbolic name for a constant appearing repeatedly in a program. Among other things, naming a constant and using its name rather than its explicit representation makes it much easier to modify your program if modification subsequently becomes necessary. To define constants, one or more constant declarations are used. Generally speaking, such declarations will have the form

(1) CONST const_name1=const_expn1,const_name2=const_expn2,..,
const_namek=const_expnk;

An example is

```
CONST pi=3.14159,two_pi=6.28318,vowels={'A','E','I','O','U'};
```

This example illustrates the following rules:

(i) Each const_namej in (1) must be a valid SETL identifier. By virtue of its appearance in (1), this identifier becomes a constant identifier, i.e. a synonym for the constant denotation, const_expnj, matched to it in (1). It retains this meaning throughout the scope of the identifier.

(ii) Each const_expnj appearing to the right of an equal sign in a declaration like (1) must be a valid constant expression. Such expressions are built out of the following:

(a) Elementary constant denotations, each of which designates an integer, a real number, or a quoted string.

(b) Constant identifiers, i.e. identifiers of constants introduced by earlier CONST declarations. For example, it is possible to write

```
CONST one=1,two=2,one_and_two={one,two};
```

This is equivalent to

```
CONST one1=1,two=2;
CONST one_and_two={1,2};
```

(c) Simple identifiers. An otherwise undeclared identifier appearing within a CONST declaration is treated as an implicitly declared string constant whose value is its capitalized name. For example, in the absence of other declarations, the declaration

```
CONST colors={red,green,blue};
```

is equivalent to

```
CONST colors={'RED','GREEN','BLUE'};
```

(d) Compound constant denotations can also appear in CONST declarations. Such denotations are built from elementary constants of the above forms (a-c) using set and tuple brackets but no other operators. This means that the constructs

```
CONST complex_thing=[{'A',1},{'B',2},{{}}];

CONST let_1='alpha',let_2='beta',let_map={['A',let_1],['B',let_2]};
```

are all legal, but that the declarations

```
CONST two_pi=2.0*3.14159;
```

and

```
CONST sixty_blanks=60*' ';
```

are invalid, since they both involve operators other than set or tuple brackets. Note also that a nested construct like

(2) `CONST number_name={[1,one],[2,two],[3,three]};`

can be used even in the absence of other declarations. Assuming that no other declarations are present, (2) is exactly equivalent to the declaration

```
CONST one='ONE',two='TWO',three='THREE';
number_name={[1,'ONE'],[2,'TWO'],[3,'THREE']};
```

(See (c) above).

(e) A constant identifier introduced by a CONST declaration retains its fixed constant meaning over the scope of the identifier (see Sections 4.2 and 9.1). This scope will be either an entire program, a program module, or a single subprocedure.

In addition to the CONST declaration form (1), the abbreviated form

(3) `CONST const_name1,...,const_namek;`

is allowed. That is, some or all of the parts '=const_expnj' appearing in (1) can be omitted. An identifier appearing with this elision in a CONST declaration is treated as an implicitly declared string constant, whose value is its capitalized name. For example,

```
CONST one,two,three;
```

is equivalent to

```
CONST one='ONE',two='TWO',three='THREE';
```

See Section XXX for an explanation of the place within programs or procedures at which CONST declaration can appear.

6.4 The ASSERT Statement

The form of an ASSERT statement is

(1) ASSERT expn;

where -expn- designates any Boolean-valued expression. To execute such a statement, the -expn- it contains is evaluated. If the resulting value is FALSE, a message of the form 'ASSERTION FAILED AT LINE XXX OF PROCEDURE YYY' is produced, and execution terminates; if TRUE, then control passes immediately to the statement following the ASSERT statement. (More precisely, a FALSE assertion will terminate execution if the 'check assertions' feature of the SETL execution-time system is switched on. Moreover, if the 'confirm assertions' feature of the SETL execution-time system is switched on, then each TRUE assertion will produce a message 'ASSERTION PASSED AT LINE XXX' OF PROCEDURE YYY'. (See the discussion in Section 8.5.1.4 of the execution-time control card parameter ASSERT).

ASSERT statements are ordinarily used in a program for one of two reasons:

(i) To document and to check logical conditions which the programmer knows to be critical for correct functioning of his program. Used in this way, ASSERT statements constitute a powerful program debugging aid. See Sections 7.2 and 7.7.1 for additional discussion of this point.

(ii) To trigger any side effects caused by evaluation of the Boolean -expn- that the statement (1) contains. Note that this -expn- can contain assignments or other subexpressions (such as existential or universal quantifiers) whose evaluation causes side effects. Evaluation of the ASSERT statement (1) will always trigger these side effects even if assertion checking is switched off). (See the discussion of control-card parameter ASSERT in Section 8.5.1.4).

Perhaps the commonest case of this second use of the ASSERT statement is in constructs of the form

```
ASSERT EXISTS x IN s|C(x);
```

This construct can be used whenever one is certain that the set {x IN

s|C(x)} is non-null, and in this case it will always give x a value such that C(x) is TRUE. A similar, somewhat more elaborate use of the ASSERT statement is shown in

```
ASSERT (EXISTS x IN s|C(x)) OR (EXISTS x IN s1|C1(x));
```

Assuming that the assertion is TRUE, execution of this statement will always set x either to an element of s for which C(x) is TRUE or set x to an element of s1 for which C1(x) is TRUE.

6.5 Macros

Macros are abbreviations that obviate the need to write similar pieces of code repeatedly ; they allow the SETL programmer to introduce and use various convenient 'shorthand' notations for constructs that are used many times in a program. Macros, like procedures, are defined once and can then be used several times.

MACROs and PROCEDUREs resemble each other in that both give ways of associating names with bodies of code text and of invoking this code when the name is mentioned. However, when a macro is mentioned in a program after having been defined, the program text which it represents is substituted directly for the invoking occurrence of the macro name; this substitution is called macro-expansion, and is to be contrasted with the detour-and-return action (see Section 4.1) triggered by a procedure invocation. That is to say, macros make use of a purely 'textual' mechanism; they simply replace the name of the macro by its definition at the point where the name appears. This means that unlike procedures (which can be invoked before their definition has been seen), macros must be defined before they are used, i.e. the definition of a macro must appear physically in a program before the macro is first used.

6.5.1 Macro Definitions

Macros in SETL are defined by using one of the following constructs:

```
(1)  MACRO m_name;
        macro-body
     ENDM;

(2)  MACRO m_name(p_name1,p_name2...p_namek);
        macro-body
     ENDM;

(3)  MACRO m_name(p_name1..,p_namek; gpname1,..,gpnamej);
        macro-body;
     ENDM;
```

The form displayed in (1) is that of a parameterless macro. The construct (2) shows that macros can have parameters. The form (3) includes generated parameters, whose purpose and use will be described below.

After a macro has been introduced by one of the above constructs, it can be invoked simply by using its name, followed by appropriate parameters. We will now examine the use of these forms, starting from the simplest one, the parameterless macro.

6.5.2 Parameterless Macros

Macros without parameters provide for the simplest kind of abbreviation: the name of such a macro simply stands for its macro body, which replaces the macro name whenever this name appears. For example, we can write:

```
(4)     MACRO countup;
            t := t+1;
            if t>limit then
                errmsg('out of bounds')
            end if
        ENDM;
```

Following the appearance of definition (4) in a program, module, or procedure, any subsequent appearance of the name -countup-, for example in the line

```
(5)     countup;
```

triggers replacement of (5) by the body of 4, i.e. by the four lines of SETL code shown above (which of course increment and test the variable t).

We note that this replacement is made by the compiler, but it is not shown in the source program listing which the computer produces. Line (5) appears in the listing as is. However, compilation proceeds as if the macro body of (4) had occurred instead of (5).

Our next example shows that a macro body need not consist of a group of statments, but can be any sequence of tokens, including sequences which are not meaningful in themselves. A macro which exploits this fact is:

```
MACRO find; ASSERT EXISTS ENDM;
```

This macro can be used as follows:

```
find x in s|c(x);
```

6.5.3 Macros with Parameters

Macros with parameters are introduced by macro definitions of the form

```
(1)     MACRO mname(pname1,...,pnamek);
            body
        ENDM;
```

Here, -mname- can by any legal SETL identifier which becomes the name of the macro introduced by (1); pname1,...,pnamek, called the formal parameters of the macro, can be any list of distinct identifiers. The -body-, known as the body of the macro, or equivalently as its macro text, can be any legal SETL text fragment.

After being introduced by a macro definition (1), the macro -mname- can be invoked simply by using its name, followed by a list of k actual arguments, at any place within a program. Suppose, to be specific, that this invoking occurrence is

(2) mname(arg1,...,argk)

Then the SETL compiler replaces the macro invocation (2) with an occurrence of the -body- of the corresponding macro-definition (1), but in this body every occurrence of a formal parameter name -pnamej- will have been replaced by an occurrence of the corresponding argument -argj-. We emphasize again that this is done by replacement of text, and not, as in the case of a PROCEDURE call, by evaluation of arguments and transmission of their values. This means that the arguments -argj- of macro-invocation need not even be complete, evaluable expressions; indeed, they can be arbitrary sequences of keywords, operator-signs, constants, or identifiers. (However, since commas are used to separate the successive arguments of a macro invocation, no argument of such an invocation can contain an imbedded comma). This gives macros a syntactic flexibility which procedures do not have, and which is sometimes useful. Suppose, for example, that we wish to print out a series of examples illustrating the use of the compound operator in SETL. This could be done directly by using the following code:

```
v:=[1,2,3,4,5];

print('Combining the components of v using the operator + gives', +/v);
print('Combining the components of v using the operator * gives', */v);

print('Combining the components of v using the operator MAX gives',
MAX/v);
 etc.
```

y using a suitable macro, we can abbreviate this repetitive code, as follows:

```
(3) MACRO print_op(opsign,op);
        print('Combining the components of v using the operator',opsign,
                    'gives',op/v)
    ENDM;

    v:=[1,2,3,4,5];

    print_op('+',+);
    print_op('*',*);
    print_op('MAX',MAX);
```

This illustrates the possibility of transmitting an isolated operator sign to a MACRO as an argument; notice that no corresponding possibility exists for PROCEDUREs.

For a second example illustrating the syntactic flexibility which sometimes justifies the use of a MACRO rather than a PROCEDURE, consider the common situation in which we need to check repeatedly for erroneous data and return some appropriate error indication if an error is detected. Suppose, to be specific, that these checks need to be made as part of some procedure, and that when an error is detected, we want the procedure to return immediately and to transmit an appropriate numerical error indication. The following MACRO is suitable for this purpose.

```
(4)     MACRO check(condition,error_no);

        IF NOT condition THEN RETURN error_no; END

        ENDM;
```

After introducing this macro, we can check for errors very simply, e.g. by writing

```
(5)     check(a<b,1);        $ error numberl
        ...
        check(f(x)/=OM,2); $ error number 2
        ... etc.
```

Note that a PROCEDURE invocation could not trigger an immediate RETURN in the same convenient way that this MACRO does.

A syntactic point to be noted is that neither the body of the MACRO (2) nor the body of (3) ends with a semicolon. This is simply because it is most natural to put the semicolon which terminates an invoked macro body after the macro invocation which triggers insertion of this body (cf. (3) and (4)). Since a substituted body replaces each macro invocation, putting a semicolon both after a macro body and after its invocation would lead (after substitution) to the (harmless) occurrence of a double semicolon. This is the stylistic reason why semicolons are omitted after the last line of the body of the macros (2) and (3).

As a final example, let us mention the oft-used macro which names the last component of a tuple:

```
MACRO top(stack);
stack(#stack);
ENDM;
```

This macro can be used in expressions as well as in assignments, for example:

```
x := top(v);
top(v) := y+1;
```

etc.

6.5.4 Macros With Generated Parameters

In addition to its ordinary parameters and arguments, macros can make use of generated parameters which play the role for macros that local variables play for procedures. To make use of this feature we write macro definitions having the form

```
(6)     MACRO mname(pname1,...,pnamek;gpname1,...,gpnamen);
            body
        ENDM;
```

The additional parameters gpname1,...,gpnamen appearing after the first semicolon in (6) but not in (1) are called generated parameters. The programmer is not supposed to supply arguments corresponding to parameters of this kind when a macro like (6) is invoked. Instead, one invokes a macro like (6) in exactly the same way as the macro (1). However, when a macro like (6) with generated parameters is invoked, the SETL compiler generates new tokens (of an artificial form that cannot be used accidentally by the programmer) and substitutes them for occurrences of the corresponding generated parameter names in the -body- of (6).

A common use of this option is to generate a supply of fresh variable names when these are required for local use within the substituted body of a macro. Suppose, for example, that we want to write a macro which tests the value of an expression e for membership in a given set s, and which returns immediately from the procedure invoking the macro in case the test e IN s fails. Suppose also that in case of failure we want to return both a numerical error indication and the value of the expression e. If we write

```
    MACRO double_check(e,error_no);
 IF e NOTIN s THEN RETURN  [error_no,e];          END;
    ENDM;
```

we would not get exactly the desired effect because when this macro is invoked, it will insert the actual argument for e in two places, which will lead to repeated evaluation of e. For example:

```
    double_check(f(y)+g(y), 15);
```

would expand as

```
     IF f(y)+g(y) NOTIN s THEN
          RETURN [15,f(y)+g(y)];
     END;
```

In order to avoid this double evaluation we can use the following macro:

```
(7)  MACRO in_check(e,error_no;temp) $ macro with generated parameter

       IF (temp:=(e)) NOTIN s THEN RETURN [error_no,temp]; END

     ENDM;
```

To invoke this macro we could, for example, write

```
(8)     in_check(t WITH:= x,1);          $ error number 1
        ...
        in_check(t WITH:= y,1);          $ error number 2
        ... etc.
```

Note that if (as in (8)) an argument expression -e-, causing some side effect, is passed to the macro (7), it becomes essential that the value of -e- should be assigned to a auxilary variable (the generated parameter -temp-) and that e should not be evaluated twice. Note also that each use of (7) will generate a new name for the parameter 'temp' so that no accidental interference will occur between invocations of this macro. Finally, note the use of a precautionary extra pair of parentheses around the occurrence of the parameter -e- in the body of (7); these parentheses ensure that the argument transmitted to the macro in place of -e- will be handled as a unit, no matter what its actual syntactic form happens to be.

6.5.5 The Lexical Scope of Macros. Macro Nesting.

The scope within which a macro will be active is determined by the context in which its definition appears. A macro name introduced by a macro definition appearing in a PROCEDURE (resp. a MODULE, PROGRAM, or LIBRARY, but outside any PROCEDURE) maintains its meaning as a macro throughout this PROCEDURE (or PROGRAM, MODULE, or LIBRARY), but not past the PROCEDURE's end. Note however that the macro can be redefined by a later macro definition appearing in the same PROCEDURE (or MODULE, etc.), or can be dropped. The way in which macros are redefined and dropped is explained in more detail below).

Macro-bodies can contain invocations of other macros; and macro- names can be transmitted to other macros as arguments. For example, suppose that we define the following two macros:

```
MACRO triple(pa);
    pa,pa,pa
ENDM;

MACRO q;
    'hello there'
ENDM;
```

Then, after expansion, the macro invocation

```
triple(q)
```

becomes

```
'hello there','hello there','hello there'
```

This example illustrates the fact that macro-expansion is outside-in and recursive. That is to say, the expansion of a given macro body may trigger the expansion of an inner macro invocation.

Macro bodies can also contain imbedded macro definitions. For example, the definition

```
(9)     MACRO def_x(pa);
            MACRO x;pa ENDM;
        ENDM;
```

is legal. An imbedded macro defintion IMD becomes active when one invokes the macro M in which IMD is imbedded, thus causing the body of M to be expanded.

As an example, note that after expansion the sequence

```
(10)    def_x('aaa');
        x x x
        def_x('bbb');
        x x x
```

becomes

```
(11)     'aaa' 'aaa' 'aaa'
         'bbb' 'bbb' 'bbb'
```

This happens in following way. The first line in (9) is expanded, and becomes the macro definition

```
MACRO x;'aaa' ENDM;
```

Then the second line of (10) is subsequently expanded. It generates the first line of (11). After this, the third line of (10) is expanded into

```
MACRO x; 'bbb' ENDM;
```

This changes the meaning of the macro x, causing the fourth line of (10) to expand into the second line of (11).

6.5.6 Dropping and Redefining Macros

If a macro is only needed over a limited portion of a program, it is possible to 'undefine' it so that the name of the macro can be used for another purpose. To erase a macro definition, one uses the following SETL construct.

DROP macrolist;

where macrolist is a list of macro names, separated by commas. Once a macro has been dropped, it is possible to give it a new definition, or to use its name for any SETL object, without confusion. For example,

```
(12)    MACRO x; print('now you see it') ENDM;
        x;
        DROP x;  $ this drops x from macro-status
        x;
```

```
MACRO x; print('now you don''t'); ENDM;
x;
```

expands into

```
(13)    print('now you see it');
        x;
        print('now you don't');
```

This follows since the first line of (12) makes x a macro equivalent to 'now you see it', but then the third line of (12) drops x from macro status, so that the fourth line of (12) carries over unchanged to become the second line of (13). The new definition of x is then seen, invoked, and expanded.

Note that the compiler will see the line

```
x;
```

as an invocation of some unspecified procedure x. If no such x exists, the program will of course not execute.

Considerably more elaborate macro features than those we have described are supported by other programming languages, especially by machine level 'assembly languages'. However, high level languages like SETL have less need for complex macro features than do lower-level languages, and thus the macro facility that SETL provides will be found adequate for the use normally made of it. Let us remark that macros, like procedures, perform the useful function of hiding low-level details, and thus help make a program more readable and more modular. The information-hiding capablity of macros is most useful when we want to 'shield' a program from possible changes in the structure of composite objects which it manipulates. The organization of a data-base is a good example. Suppose that a library catalog is to be built. Each book has an entry in the catalog, which includes the title, author, date of publication, subject, and library of Congress number. The catalog itself can be structured, let us say as a map whose domain is the set of call numbers, and whose range is a set of tuples of length 5, containing the above information. In this situation, we may find it appropriate to write

```
MACRO title(call_number)
    catalog(call_number)(1)
ENDM;

MACRO author(call_number)
    catalog(call_number)(2)
ENDM;

    .
    .
    .
```

thereby hiding the tuple structure of the data from its user. This allows us to write:

```
if author(x) = 'Barth'...
```

rather than having to recall that the author is stored in the second component of an element of the range of the catalog, etc.

6.6 Programming Examples

In this section we collect various programming examples which illustrate the use of the SETL features descibed in this section.

6.6.1 Iteration Macros

Frequently one will be given a map (or programmed function) and an initial element x, and will need to iterate over all the elements y= x, f(x), f(f(x)),..., performing some operation repeatedly until an OM element terminating the itertion is reached. Iterations of this kind can be written as

```
(1)     y:= x

        (WHILE y/= OM)
            body of iteration
            y:= f(y);
        END;
```

However, if a program uses many iterations of this kind, it may be worth introducing a macro to abbreviate them. Using SETL's generalized loop construct such a macro can be written as

```
    MACRO ORBIT(y,x,f);
        INIT y:= x; WHILE y/=OM STEP y:= f(y)
    ENDM;
```

This macro enables us to write the loop M1 as

```
(1A)     (ORBIT(y,x,f)) body of iteration END;
```

Note that the iterator introduced in this way can also be used in setformers and tuple-formers, e.g. we can write

```
    +/[e(y): ORBIT(y,x,f)]
```

to form the sum e(x) + e(f(x)) + ..., which includes all terms e(x), e(f(x)),... up to the point at which f first becomes undefined.

Another commonly occurring but somewhat more complex case is that in which a map f is multi-valued, and we wish to generate all elements y belonging to any sequence of elements z1 ,z2,...,zn starting with x=z1 such that [zi,z(i+1)] is a member of f for all i in [1..n-1]. (In mathematics, this set is called the transitive closure of {x} relative to f). To iterate over the elements of this transitive closure (in a somewhat unpredictable order), we can use the following loop, which makes use of two auxiliary variables -to_process- and -seen_already-

```
(2)     s:= {x}

          to_process:= seen_already:= s;

          (WHILE to_process/= { })
              body of iteration
              y FROM to_process;
              to_process +:= f{y}-seen_already;
              seen_already +:= f{y};
          ENDM;
```

This loop can be abbreviated by introducing the following macro

```
    MACRO TRANS_ORBIT(y,s,f; to_process,seen_already);
        INIT to_process:= seen_already:= s;
        STEP y FROM to_process;
            to_process +:= f{y}-seen_already
        WHILE to_process/= { }
        seen_already +:= f{y};
   ENDM;
```

Using this macro, the loop (2) can be written as

```
(2A)    (TRANS_ORBIT(y,{x},f))  $ y iterates over all the elements
                                $ of the transitive closure of f
            body of iteration
        END;
```

This iterator can also be used in setformers, tuple-formers, etc. For example, we can write

```
    +/[e(y): TRANS_ORBIT(y,s,f)]
```

to sum the expression e over all the points belonging to the transitive closure of s relative to f.

6.7 Exercises

Ex.1 Write a constant map which sends each month of the year into the number of days in the month. Assume that February always has 28 days.

Ex. 2 The code

```
    sum:= 0;
    (FOR c=t(i)) sum +:= c; END
```

sums the components of the tuple t. Into this loop insert an ASSERT statement which relates the value of the variable sum to the integer i. Execute this code and verify that the asserted assertion is always TRUE. How much slower do you expect the inserted ASSERT statement to make this loop run?

$

CHAPTER 7

PROGRAM DEVELOPMENT, TESTING, AND DEBUGGING

As noted in Section I.2, the normal stages of a program's life-cycle are

(i) Initial conception, formulation of requirements.

(ii) Overall design of a progam that will meet these requirements.

(iii) Detailed design and coding.

(iv) Program review, with rework and extension as needed to clarify, simplify, or improve efficiency.

(v) Development of a test plan; testing and debugging; removal of errors, and retest.

(vi) Operational use of program.

(vii) Enchancement and repair during continuing operational use.

(viii) Retirement.

This chapter discusses various key aspects of this program life-cycle, providing hints that aim to help the inexperienced programmer to cope effectively with the pragmatic problems normally associated with program design, debugging, and maintenance. Proper understanding of these issues can improve the overall effectiveness and quality of your work.

Chapter Table of Contents:

7.1 Bugs: how to minimize them.

Any small error affecting the behavior of a program is called a bug. Bugs are inevitable, but a few cardinal rules can help minimize the degree to which they infest your programs.

(i) Know that they will occur. Since any small error, i.e. forgetting a line, typing '-' where '+' is meant, misspelling an identifier or keyword, mis-parenthesising an expression, will cause a bug, you must train and discipline yourself to higher levels of logical and typographical accuracy in programming than are required in any other human activity. Be suspicious. Program defensively. Check your programs scrupulously for syntactic and logical correctness, several times if necessary, before you try to run them. If in doubt as to the meaning of any operation or programming language construct, look it up.

(ii) When bugs occur, your problem is to locate, recognise, and remove them. Bugs cannot be located unless you know the programming language with which you are working well enough to recognise problems when you are looking at them. Bugs cannot be eliminated until you have understood them well enough to know why and how they cause the faults that betray their presence. Finding bugs, like finding needles in a haystack, calls for systematic sifting, for careful detective work. A program is a delicate piece of machinery, and it is simple folly to think that you can make it work by kicking it hard in some random way to make its pieces fall into place. Because they involve many submechanisms, all of which must interface correctly if they are to work together properly, programs, like elaborate combination locks, require careful analysis and attentive sensing of their hidden internals when they need repair. The novice who tries to fix a malfunctioning program without fully understanding the way in which it is working is attempting a task that is far less hopeful than that faced by someone who tries to open an unfamililiar safe without understanding its workings or combination. 'The sequence 33-8-19-27 doesn't work? Then I'll try 23-92-69-46. This doesn't work either? Then maybe 17-51-85-34 will be luckier.' A student who allows himself to be drawn into of this sort of thoughtless, random attempt to diagnose or repair a program will inevitably find that his efforts drag on unsuccessfully, not only till the end of the term or year, but until the end of the solar system, without revealing anything. What is needed instead is a systematic, analytic approach.

(iii) Though programs are almost never entirely bug-free, observance of the rules of good programming style can reduce the density of bugs in your initial program drafts and allow bugs to be found more quickly once testing of your program begins. Finding the right approach to the programming task that confronts you, the right style in which to start writing the code that you need, is of prime importance. To find this 'right approach' requires careful consideration of the logical structure of your programming task, with the aim of defining a collection of intuitively transparent operations which work well together and can be used to accomplish this task in as straightforward a way as possible. Code should impress by its clarity, naturalness, and inevitability, all of which make avoidance and exposure of bugs easier, rather than by obscure trickery and impenetrable cleverness. Programs that achieve brevity without sacrificing clarity are most desirable, since lines of code that you never need to write will never contain bugs. Effective brevity is attained by a correct choice of intermediate operations and by systematic use of these operations to produce the program you require. SETL is in itself a powerful programming language, but especialy for larger, more complex applications it may be well to program by first inventing a still more powerful language specially adapted to your intended application. Then your initial program draft can be written in this (possibly unimplemented) language, after which it can be transcribed mechanically into SETL to make it executable. In this sort of approach, the primitives of your invented language will become the second-level procedures and macros of your SETL code. By using an auxiliary language in this way and by handling its transcription into SETL in as mechanical a style as possible, valuable protection against error is gained. See Sections 4.2, 9.1 below for a discussion of related issues, and Section xxx for an extended example of what is meant here.

(iv) Careful program documentation also serves to expose and eliminate bugs. Good documentation will add an important degree of redundancy to your program. Your code expresses your intent in one way and your comments express the same intent in another. Discrepancies between the two indicate the presence of bugs. Carefully thought-out comments should be added to a program as soon as the code is written. Some comments will in fact be written before the code to which they refer, in order to guide composition of the code. Any additional comments needed to make documentation complete should be added to the code while it is still 'fresh'; this creates an opportunity to review the code, checking it for logical faults. After the whole text, code plus comments, has been constructed and put into proper format, it should be left to 'cool' for a few hours or days, after which it should be reviewed attentively and suspiciously. Such a 'cooling off period' will dispel some of the initial misapprehensions which may have crept into a code, and thus will allow various systematic errors to be corrected.

(v) As has been said, brevity in coding is desirable, but this should be the kind of brevity that flows naturally from an effective overall approach to the programming task at hand, not the undesirable brevity which comes from stinting redundancy (e.g., by using short, un-mnemonic, variable names.) Use the features of the SETL language vigorously and eliminate clumsy circumlocations where direct modes of expresion exist; but avoid obscure tricks even even where these gain brevity.

(vi) Certain constructions, for example those which perform elementary

arithmetic computations to determine positions in strings and tuples (for which 'off-by-one' errors can easily occur) are bug-prone and need to be approached with caution. For example, what is the length of a string s(i..j), is it j-i, j-i+1, or j-i-1? To ensure that s(i..j) is exactly k characters long, what value do we give j: i+k-1, or i+k+1? Learn to recognise these trouble spots, double-check item when preparing your code, and surround them with ASSERT checks when you do use them. For example, if you write

```
ASSERT  s(i..j)=j-i;
```

immediately before proceeding on this assumption, your error will be pinpointed immediately; if you omit this check, you may have to find your way back to this error from some obscure symptom. A related idea is to introduce, and use, a collection of standard macros to handle these touchy situations in ways that are more instructive. For example, by introducing the macros

```
MACRO len_from(i,j); j-i+1 ENDM;
MACRO make_len(i,k); i..i+k-1 ENDM;
```

we can accurately extract a string of length k from (s) starting at character position i by writing

```
s(make_len(i,k))
```

and can evaluate the length of s(i..j) by writing len_from(i,j)

(vii) As Donald Knuth has remarked, premature optimisation is the root of all evil in programming. Compulsive (and ultimately ineffective) attempts to gain minor efficiency advantages often complicate programs and introduce bugs into them. As you compose a program, remember that substantial efficiency advantages will be gained globally by choice of effective algorithms, not locally by complicating seizure of minor advantages.

(viii) Prescreening routines, for example routines which examine a program for tokens (such as RETRN) which are likely to be mispellings of other, more frequently used, variable names or keywords may be available to you. Grow accustomed to using these bug-finding aids (like RETURN in this case), as well as any other available compile-time debugging aids based on more sophisticated global program analysis routines.

(ix) Your program test plan should begin to be developed as your program is being written, and a substantial portion of the collection of test-and debug-oriented PRINT and ASSERT statements that you will use to test your program should be composed and entered as soon as the first draft of the program begins to approach completion. Early attention to your test plan will serve to pinpoint complex program sections that require careful testing. These are also the sections whose logic needs to be inspected most closely before testing begins. See Sections 7.4 and 7.7 for additional discussion of this point.

7.2 Finding Bugs

Even, alas, if you are very systematic and professional, some bugs will creep into your program, and the problem will then be to find and fix them. The following remarks should help you learn how to do this effectively. Debugging always starts with evidence that a program error has occurred somewhere during a program run. The problem in debugging is to work one's way back, from the visible symptom first noticed, to the underlying error. The errors one is looking for can be called the error sources or primal anomalies: These are the first (incorrectly written) operations or statements which get correct data from what has gone before them, but pass data that is no longer correct to what comes after them. They are the instructions at which your program first 'runs off the rails'. The initial evidence of error that you see may relate only indirectly to these primary error sources. The difficulty of finding the erroneous statements is complicated by the fact that the full history of an extensive computation comprises a vast mass of data, impossible to survey comprehensively. In debugging you must therefore aim to explore as narrow a path as possible, while still finding your way back to one or more primal anomalies.

A good first step, but one that should not be allowed to hold you up too long, is to look closely at whatever fragments of correct output have been produced. If little or no output is correct, then your program may have failed before even the first PRINT statement was executed. This hint may help you narrow the bug hunt. On the other hand, if some output is correct, then the program was probably functioning correctly till some point past the statement which produced the last correct output. Find the point in your program at which this output was produced, and see what comes before and after it. Again, this may narrow the hunt. Examine the erroneous output carefully and try to see if its logical pattern reminds you of any particular section of your program. This also can sometimes yield useful hints concerning the likely location of the bug, especially if different parts of your output data are produced by recognisably different sections of code. If certain items of output that you expect are missing, try to see what evidence there is that all the code that you expected to execute did actually execute: remember than unanticipated data may have caused your program to follow an unexpected path through its code, so that it may have bypassed, or may never have reached, the code sections which were supposed to have produced the output which you are surprised not to see. Evidence of this general kind, analysed, will in favorable cases point the finger of suspicion at certain narrow program sections. However, in less favorable cases, the available evidence will be ambiguous. In this case, you will need to generate more extensive traces and dumps. This can be done in one of two ways:

(a) By inserting additional PRINT statements into your program, to make it print out something of a 'motion picture' of what has happened.

(b) By inserting various other checks, especially ASSERT statements, which check assumptions on which your program depends, but which you are afraid might be failing.

Sections 8.5.1.4 below will have more to say about technique (b), which is related to the general issue of formal program validation. The following more pragmatic hints will help you to apply this technique effectively. It is particularly important to place ASSERT statements in sections of code known to involve delicate constructions, especially if (as in the case of the 'off by 1' bugs considered in the last section) the necessary checks are simple. Since the correct functioning of a program often hinges upon the assumption that key variables will change in a consistent way as iterative execution proceeds (for example, always increasing or always decreasing) it can be useful to save the last previous value of each significant variable -var- and to write checks which compare the last previous value of -var- with its current value. This can be done by introducing an auxiliary variable -last_var- for each -var-, and writing an assignment

```
last_var:= var;
```

whenever it is desirable to save the last value of -var-. Then checks like

```
ASSERT var=last_var;
ASSERT var/=last_var;
ASSERT last_var=OM OR var*last_var={ } AND var/=last_var;
```

etc. will all prove useful.

It may be useful to check an assertion the first few times it is encountered, but not subsequently. (If this is done, and you select the option (see Section 7.2) which prints a confirmation message each time an assertion is checked, you should be able to tell that your program is following its expected path.) The following macro is convenient for this purpose:

```
MACRO ASRT(n;temp);
    temp ?:= (n+1); ASSERT (temp -:= 1)=0 OR
ENDM;
```

If we use this macro, and for example write

```
ASRT(3) C;
```

then the condition C will be checked the first 3 times that it encountered. (Look at this macro definition carefully, and make sure you understand the way in which the dangling OR at the end of it controls the execution of the expression C which we want to ASSERT a few times.)

Ultimately, however, the problem with a purely assertion-based debugging technique is that it is not easy to formulate the necessary checks comprehensively enough to make it unlikely that a bug (which probably relates to something that has been overlooked) can slip through.

Hence one must often fall back on on method (a), which generates additional raw evidence for inspection. The problem in using this method is to avoid burying yourself in too voluminous a trace of the thousands, or even millions, of events that take place as a program executes. To avoid this danger a carefully planned sequence of probes is necessary. A good idea is to resurvey your program, mentally list its main subphases, and determine all the data objects which each phase passes to the next phase.

If your program has been well designed, there should not be too many of these objects, and then it is reasonable to print them out for inspection. Before inspecting this information, review the logic of your program, and make sure you know just what features you expect to find in values of the variables that you have printed. Try to be aware of every feature on which any part of your program depends. Then check the actual data. If the data printed at the end of a phase looks correct in every detail, then this phase is probably correct. If something strange looking appears in the data produced by a given phase, while the data supplied to this phase looks correct, then there is probably something wrong with the code of this phase.

When this stage of debugging is reached, you will at least have determined which of the several phases of your code contains the error for which you are hunting. At this point, it is a good idea to think over all the evidence that you have examined, and see if any compelling picture of the problem seems to suggest itself. Sometimes the fact that the offending phase has now been located removes enough confusion for the difficulty to be guessed quickly. If not, you will have to carry your tracing to a more detailed level. This is a matter of inserting PRINT and ASSERT statements more densely into the offending phase, in order to locate the particular subphase that contains the error. As before, this is the subphase to which good data is being supplied, but which is seen to pass bad data along to its successor subphase.

(c) Once the bug location has been pinned down to a program section roughly a dozen lines long, review the logic of these lines. Read them very closely, looking for some misunderstanding which could have produced the anomalous data which you know that this section has generated. Try again to correlate data features with the operations responsible for producing these features. If this doesn't work, take the data supplied to the erroneous subphase, and try to trace the way that the subphase will act on this data, by hand, step by step, until you spot some error.

(d) In most cases, these steps will find the bug without too irritating an expenditure of effort. However, in the stubbornest, fortunately rare, cases the problem for which you are hunting may still elude clear identification. In these particularly resistant cases one of three causes may be at fault:

(i) If the algorithm which you are using is complex, you may have misunderstood its logic. It may be that no single line of your code is wrong: rather, its overall pattern may be subtly wrong, causing it to produce the output you see, rather than the results you wrongly expected it to generate. Global logic errors of this sort are often quite confusing. If you come to suspect that a problem of this sort has occurred, you should reason once more through the structure of your program, trying to convince yourself by careful analysis that it is logically sound. Section 7.7 below describes the formal rules that underlie reasoning of this sort.

(ii) There may be nothing wrong: you may simply have misunderstood what output your program was supposed to produce. Or you may have been looking at the wrong phase of a program which really does contain a bug, because you thought that the output of this phase showed some error, while in reality the bug was elsewhere. Or you may not have been running the program you thought you were running, or the version of the program you thought you were running, or your program may have been reading input data different from that assumed. In such case, take a few minutes to cool off, review the whole situation, including the logic of your program, once more, and start over.

(iii) Your problem may be caused by a true 'system bug', that is, an error, not in your program, but in one of the many layers of prepackaged software, including the SETL compiler, execution-time library, or operating system under which you are running. Concerning bugs of this kind we can say the following:

(iii.a) Don't be too quick to suspect them. Though such problems do crop up from time to time, they are much rarer than errors in your newly-written programs. Remember that dozens of people are using the same software systems that you are, and that if the problem afflicting you is a system-level problem, it would affect all of these people. Before you become willing to blame your problem on anything other than an elusive fault in your own program, you should always have examined your program with great care, located a section just a few lines long which you can be sure is receiving correct input (because you have printed and inspected its input) and producing bad output (again, you must have printed and inspected this output.) Finally, meticulous examination of these few lines, with review of the definition of all the operations these lines involve, of the parenthesisation of those lines, and of any applicable rules of operator precedence must give you 'courtroom' evidence that the system is not performing according to its specifications. At this point you are almost (but still not quite) in postion to report a system problem to the expert in charge of maintaining your copy of the SETL system (or of the operating system within which the SETL system runs.) Before doing so, however, you should try to simplify the evidence still further, isolating the malfunctioning lines into a malfunctioning program just a few lines long, and then paring this program down still further if possible, ideally to the point at which it contains just three lines: an assignment initialising a very few variables, a single line which obviously does not function as it should, and a print statement which confirms the fact that this line has failed to act in the manner demanded by the rules of SETL. If the system problem which you think lies at the root of your troubles disappears somewhere during this sequence of steps, the cause of your difficulties may not be a system problem at all, but an error or misunderstanding on your part, which your attempts to locate the suspected 'systems problem' may have clarified. In this case, chastened, you should return to your original program, fix the error in it, and continue your debugging. If, however, you do succeed in creating a very short program which gives unmistakable evidence of system malfunction, you should transmit a complete, clean copy of this program to a system expert. This should be accompanied by a clear explanation of the problem you have pinned down. He will then take steps to fix the SETL system, or to have it fixed.

Note that problems in the SETL system, like problems in your own programs, are most likely to concern marginal, rarely exercised cases, e.g. treatment of null-sets, null-tuples, null-strings, etc. Though the system has been in use for a few years and has been tested fairly extensively, exhaustive testing of so complex a system is simply not possible. (See Section 7.4.1 for a discussion of some of the issues involved in attempts to test programs comprehensively.)

There is a few cases in which it is reasonable to jump a little more rapidly to the conclusion that a system bug is affecting you. One is the case in which two runs of absolutely identical programs and data yield different results. Another is the case in which insertion into your program of a statement which is harmless by definition changes the behavior of the program significantly. For example, if insertion of a -print statement- changes your program's flow of control, something is obviously amiss at the system level. This may be evidence that can be reported to an expert immediately: but see the caution extended in (f) below.

(e) It should be clear from what has been said that one of the very first things you will want to trace when you start to analyse a malfunctioning program is the input data it is reading. Always 'echo' this data by printing it out immediately after it is read. Your input data may not be what you think it is, or you may be reading it incorrectly.

(f) Especially if a difficult bug is being pursued, debugging as an activity tends to create an atmosphere of confusion, which grows like a thundercloud as the mind struggles to free itself from the misapprehension which first allowed the bug to slip in. Particularly difficult bugs sometimes make one feel that one is going insane, since the laws of logic seem to be breaking down. To combat this perilous confusion, you must maintain a very deliberate, step-at-a-time, and above all skeptical, attiude while you are debugging. Verify the situation at every turn; look at what really is in your source text rather than trying to remember what was there: print out a record of what your program is really doing rather than guessing what is going on. Inexperienced student programmers often come to advisors with old versions of programs that they are trying to debug, claiming that 'I ran this program on Tuesday, and I made two or three changes that I am sure are harmless, and now it does't work.' A more experienced programmer, who knows that the only valid evidence to work from is a current, single, untorn listing showing program and output unmistakably together, will only laugh at this.

To reduce the level of your own confusion, it is sometimes helpful to work over your problem with a friend, trying to explain what is going on, and reviewing salient parts of the logic of your program with him, till he begins to understand it. A more expert consultant will often be able to spot the trouble that you have missed, but even if your 'consultant' is less expert than you yourself, you will often find that the very act of explaining the problem lets you spot what is wrong.

(g) Even when a program has once begun to function (and often even when it has been used successfully and intensively over a considerable

period), it may still contain bugs, which can lurk within sections of code which are rarely, perhaps almost never, exercised. For this reason, code inserted for debugging should generally not be removed once the bug is found. Don't throw away your crutches: it may become necessary to debug the same program again! Instead of removing debug code, you can 'comment it out' by inserting a dollar sign at the start of each line inserted for debugging. (Only inserted lines that never generated any evidence useful for debugging should be wholly removed.) Another technique, particularly useful during extended development and debugging of large programs, is to make the most valuable debug prints and checks 'conditional', by including them in IF-statements containing conditions which are normally false but can be turned on by supplying control card parameters. (See Section X for a discussion of control card parameters.) If this is done, it becomes possible to examine the inner working of a malfunctioning program quickly, without having to recompile it all.

7.3 A checklist of common bugs.

Certain bugs occur quite frequently, and the experienced programmer learns to recognize their characteristic symptoms. Here is a checklist of commonly occuring bugs, with some indication of the symptoms they are likely to produce. We only list bugs that would pass through compilation undetected.

Bug	Likely SETL Symptom
Variable not given any initial value	'Illegal data-type' error
Incorrect termination of a loop (e.g. count off by 1)	Missing items in data collections, sums too small if loop terminates too soon; 'Illegal data-type' error if loop terminates too late
Incorrect limits in string and tuple slices (e.g. count off by 1)	(Similar to incorrect loop termination)
Incorrectly structured WHILE-loop conditions or bodies, or incorrect initial conditions in WHILE-loops	Failure of program to terminate
Incorrect treatment of initial cases in recursions, or bad procedure calls	Failure of program to terminate. Possible memory overflow. Other effects can sometimes be very subtle
Omission of QUIT or CONTINUE statement	Program 'runs on' into code not intended for execution. Effects can be quite subtle.
Mispelled variables, e.g. AO for A0, B1 for Bl, c1 for ci	(Like uninitialised variable)
Reading unexpected data	'Illegal data-type' error, possibly no output
Unanticipated characters encountered by string-scan operations	Failure of program to terminate
Not resetting a counter or accumulator	Effects can be subtle. (see 'Incorrect loop termination')
Failure to set a program switch	Effects can be subtle
Parameters out of order in procedure call	'Illegal data-type' error (generally easy to find)
Shared global variable unexpectedly modified by invoked procedure	Efforts can be very subtle, and particularly hard to

or function	find if a function is involved
Variable inadventently modified by assignment to a variable intended to be different but having the same name.	If no data-type error is caused, effects can be subtle
Complex, incorrect combination of Boolean conditions	Effects can be very subtle
Mis_parenthesisation of logical or arithmetic errors, misunderstanding of precedence rules.	Effects can be very subtle
Variables out of order in READ statement	Illegal data-type error (generally easy to find)
Read operations of program inconcistent with data actually present in input file	Illegal data-type error (generally easy to find)
Target of an assignment statement misspelled	Effects can be very subtle

7.4 Program Testing

Debugging is the process of searching for the exact location of a program error when you know that some error is definetely there. Testing is the systematic exercise of a program which you believe might be correct, in an effort to see whether bugs are really absent. If testing shows a bug, debugging starts again. If your tests are not systematically designed, then bugs may go undetected even if present in your program. All one knows about a poorly tested program is that it works in the few cases for which it has been tried; it may fail in many others.

Test design is as important a part of program development as the choice of algorithms and data structures. Development of a test plan should begin while a program is being written. A procedure which is hard to test is apt to be bug-prone, and should be simplified if possible. By keeping testability in mind, you will avoid unnecessarily complex constructions, and produce cleaner, sounder code.

Testing falls into three distinguishable phases, which we will call first stage testing, second stage or quality assurance testing, and maintainance or regression testing. First stage testing begins as soon as a program is complete enough for execution to be possible. Its hypothesis is that bugs are present in sufficient numbers to prevent much of the program from working at all. During first-stage testing, one aims to make the main facilities of the program being debugged operable by finding and removing bugs quickly. Quality assurance testing begins where first stage testing ends. It assumes that a few obscure bugs remain in the program to be tested, and aims to test systematically enough to smoke them out. Maintainance testing aims to ensure that new bugs are not introduced into old programs during their extension and repair.

First stage testing

First stage testing should work through a program 'bottom up', first testing the bottom-level procedures (or code paragraphs) which implement the basic operations used by the rest of the program. Once the code realising these operations has been checked and found to be operable, the testing process will focus on intermediate-level procedures, and once these have been checked one will begin testing the program's main capabilities. Attempts to short-circuit this systematic, level-by-level test procedure by jumping directly to tests of higher program levels are more apt to waste time than to speed things up, since the lower-level causes of higher-level failures will then be hard to understand. For systematic testing, test input will need to be prepared for each procedure to be tested; this should be designed to make the output produced easy to inspect. If any of the procedures being tested make use of difficult or obscure data structures, it may be necessary to develop auxiliary output procedures which print these data structures in formats which clarify and emphasize their logical meaning. When such procedures become necessary, they should be written and tested immediately.

Perhaps because realism might conduce to suicide, programmers are generally over-optimistic concerning the likelihood that a program that they have just written will work right away. Careful preparation of a first stage test plan serves to counteract this common illusion; the more realistic attitude thereby engendered encourages more careful initial program inspection, and this often reduces the number of bugs present when first stage testing begins. This is why programs developed cautiously often become operational quickly, whereas programs developed in too optimistic a frame of mind often begin to work only after frustrating and totally unexpected delays.

An effective way of organising tests is to group them into a single procedure called -test_prog(s)-, whose one parameter s is a string consisting of test names separated by asterisks. The -test_prog- can then have the following structure:

```
PROC test_prog(s); $ skeleton of test procedure
(WHILE s/='')
    IF SPAN(s,'*')/=0M THEN CONTINUE; END;
    IF (tn:=BREAK(s,'*'))=0M THEN tn:=s,s=''; END;
    PRINT('Beginning Test',tn);
    CASE tn OF
...
(put sequence of named tests here)
...
ELSE
    print('Unnown test name');
END CASE;
END WHILE;
END proc test_prog;
```

To trigger a sequence of tests named test_4, test_2, etc. one has only to write something like test_prog('test_1*test_2*...'). Later, when first stage testing is complete, this call, and the -test_prog- procedure, can be left in the program P that has been tested, but the argument of the test_prog call can be changed so that it reads test_prog(getspp('TESTS=/')) (see Section 8.5 for an account of the -getspp- library procedure.) If this is done, no tests will be executed unless P is invoked with a control card parameter of the form TESTS=test1*test2...*testn, in which case the named tests will be performed. This approach makes it easy to retest a program in which unexpected trouble has developed. Of course, the test facilites available should be carefully documented at the start of the -test_prog- procedure.

Especially when a long program P is being developed, it may be desirable to begin testing before all parts of P have been coded (or even designed in detail. Of course, such a approach will be reasonable only if P has a sound, hihgly modular overall design, and only if the missing sections of P have been designed in enough detail so that you can be sure that no inconsistency will develop when they are designed and coded in detail.) This mode of organization of development and testing is sometimes called 'top-down' testing. It has the advantage of allowing testing and development to proceed in parallel. A related advantage is to provide particularly early confirmation of overall design soundness, or, if a design

proves to be unsound (say, in terms of 'human factors', i.e. useability), to give early warning of trouble.

If a top-down approach to development and testing is taken it will be found useful to provide a standard, multiparameter library routine having the name MISSING_SECTION. Then parts of your program that have not yet been coded can be replaced by invocations

```
ISSING_SECTION(name_of_missing_section);
```

where the string-valued parameter -name_of_missing_section- should assign the missing section a name that can be printed. The MISSING-SECTION procedure should also allow optional additional parameters, so that it can be invoked by

```
MISSING_SECTION('name_of_missing','p1 p2... pk',p1,...,pk);
```

where 'p1', 'p2',.. name various parameters with which the missing section would have to deal or which might explain why it was (perhaps unexpectedly) invoked.

7.4.1 Quality assurance testing

Second stage (or 'quality assurance') testing should aim to exercise a program comprehensively, in at least the following senses:

(i) It is obvious that parts of your program that have never been executed during debugging may well contain unrecognised errors. The battery of tests you develop should therefore force every line of your program to be executed at least once.

(ii) If your program branches on a Boolean condtion, then you will want to supply at least one test case in which the condition evaluates to TRUE, and another in which the condition evaluates to FALSE.

(iii) Improper treatment of extreme values is a common cause of program failure. A program may work for nonnull sets, tuples, or strings, but not for the corresponding null cases; for positive integers n but not if n=0; for integers less than the length of some string, but not for integers equal to this length, etc. It may work when a WHILE or a FORALL loop which it contains is entered, but fail if the loop is bypassed entirely .

In preparing a comprehensive collection of tests, you will therefore need to survey your program systematically, listing marginal situations of this kind as exhaustively as you can; then at least one test that will force each logically possible situation to occur should be prepared.

(iv) Once a list of all procedures, loops, branches, code sections, and marginal cases to be tested has been collected and a comprehensive set of tests has been developed, it may be worth preparing a formal test coverage matrix which shows which tests exercise each program feature. A chart of this kind makes it easier to spot cases that have never been tested. It can also help to select tests to be run during regression testing (see below), and can help to pinpoint program sections to be examined when a test fails. Such a chart will also make it easier to avoid running too many tests all of which exercise the same limited group of program features but never use others. Note that, if regarded as a kind of test, 'production' use of a program is subject to this objection, i.e. daily use of a program often exercises only a limited subset of its features. This is why programs that have been in heavy use for extended periods sometimes fail when their usage pattern changes significantly.

(v) Compilers sometimes include features which make it easier to determine the coverage provided by a family of tests. For example, it may be possible to generate a listing of all program statements executed during a sequence of runs, of all branches taken, of all procedures invoked, etc. The SETL measurement facility described in Appendix XXX is not untypical of 'profiling' facilities of this kind. You will want to familiarise yourself with these facilities, since they can help assure that the test-sets you develop for your programs are adequate.

(vi) Once developed, test sets become an important adjunct to the programs that they test. Such test sets should therefore be organised in a manner which facilities their long-term maintainance and re-use. The tests which are available, and the coverage they provide, should be adequately documented.

(vii) Programmers often find it hard to bring sufficient enthusiasm to the task of systematically rooting obscure bugs out of code that they themselves have written. In part, this is a matter of over-optimism; in part, a result of the mental fatigue which tends to set in at the end of a lengthy code development; in part, a consequence of the difficulty of overcoming the very mind-set which introduced an error in the first place. For all these reasons, it is good practice to make testing of large programs the responsibility of a quality assurance group independent of the development group that produced these programs. If this is done, then, knowing that an independent group of programmers will probe their work systematically to find shortcomings, the original development group will be encouraged to simplify their product so as to improve its reliability.

Even where resources do not permit fully independent organization of the activity of program testing, it is well to ensure that every line of a complex code is read and understood by at least two programmers, each of whom will be able to spot problems and suggest tests that the other might have overlooked.

7.4.2 Regression Testing

Regression testing is testing routinely applied whenever a previously working program is amended, to ensure that newly introduced code has not caused new errors. Tests which will be used in this way should be written so as to be self-checking, i.e. to produce little or no output if they have run correctly, but to produce copious output pinpointing a problem as closely as possible when an error is detected. This can be done by organising the tests so that they perform various calculations, always in at least two different but logically equivalent ways. If these paired computations produce the same result, then either no output, or a simple message 'TEST xxx PASSED', should be printed, but if a discrepancy is detected output which shows the discrepancy and displays the values of all variables related to the discrepancy should be printed.

If a chart has been prepared showing the program features exercised by each test (see iv above) it can be used when a test fails to suggest what part of the program should be examined first to find the cause of failure. If some one of these program sections has just been changed, it will of course come under immediate suspicion.

7.5. Analysis of program efficiency

7.5.1. Efficiency of some of the basic SETL operations; estimating the execution time of loops.

It is very easy to use SETL (or any other programming language) to write programs which would take years, or even hundreds or thousands of years, to finish executing. Consider, for example, the code fragment

```
(1)     sum:=0;

        (FOR i IN [1..1000])
          (FOR j IN [1..2000])
             (FOR k IN [1..3000])
               (FOR l IN [1..4000])

                  sum +:=(2*i*i*i+j*j*j+k*k+l);

               END FOR l;
             END for k;
           END for j;
         END for i;
```

In this code, for each successive value i, the variable j iterates over 2000 different values; for each value of i and j, the variable k iterates over 3000 values; and for each value of i,j, and k, the variable l iterates over 4000 values. Thus, all in all, the innermost statement of the code fragment (1) will be executed 1000x2000x3000x4000 times, i.e. 240 billion times. This statement involves 6 multiplications and 4 additions, so that at least 2.4 trillion elementary arithmetic operations are required to execute the code (1). Even on a computer capable of executing a million arithmetic operations per second (a fairly typical performance figure nowadays) and even if the code (1) were written in a programming language capable of exploiting this raw arithmetic capability to the utmost, 2.4 million seconds would be needed to execute the code (4). Since an hour is about 4,000 seconds, this is about 600 hours, i.e. about 24 days. However, since SETL (which pays a price in efficiency for its very high level) is roughly 30 times less efficient than this, execution of the SETL code (1) would require roughly two continuous years of computer time. This makes it quite clear that in writing SETL programs one needs some way of estimating the computational resource which will be consumed to execute the code that one sets down.

At least for the most straightforward programs, this is not hard to do. Elementary operations on integers and real numbers can be considered to take one nominal 'unit' of time. (Depending on the speed of your computer and the quality of the SETL implementation that you are usng, this 'unit', in terms of which we will state all our other timing estimates, could be anything from a millionth to a ten thousandth of a second.) Any simple assignment operation x:=y should take roughly one unit of time, as should a tuple indexing operation t(i). If f is a map, than the map indexing operation f(i) is somewhat slower, say roughly five times as slow. The set membership test x IN s also takes roughly five time units. (See Section 10.2.)

Basic opperations on composite objects, for example set union, difference, and intersection, also tuple and string concateration, take a time proportional to the size of the objects involved. For example, forming the concatenation t1+t2 of two tuples (or strings) takes a time proportional to the sum of the lengths of t1 and t2, since all the components (or characters) of both t1 and t2 must be moved into the concatenated tuple that is being formed; at the elementary 'machine' level which underlies the SETL implementation, these components are moved one at a time. Generally similar remarks apply to the operation of forming the union of two sets, but here the situation is actually somewhat more complicated, and we postpone its detailed discussion to Section XXX.

Iteration over a set, as for example in

```
(FOR x IN s)...END;
```

or over a map, tuple, or string, as in

```
(For x=t(i))...END;
```

produces set elements (or map values, tuple components, or string characters) at a rate of one per cycle. Essentially the same remark applies to numerical iterators, like those in (1) above. Hence to estimate the time required to execute a loop, we have only to multiply the number of times the loop will be executed by the (average) time that it will take to execute the body of the loop. An obvious generalisation of this rule applies to imbedded loops: if one FOR loop is imbedded within another, then the time required to execute the outer loop is the product of the number of times it will be executed, times the (average) number of times the imbedded loop will be executed, times the time required to execute the body of the imbedded loop. For example, the double loop

```
(FOR i IN [1..1000], j IN [1..i])...END;
```

will execute in a time roughly equal to 1000x500 multiplied by the amount of time required to execute the loop body, since the (implicitly) imbedded loop over j will execute an average of 500 times for each successive value of i (This number 500 is halfway between the number 1 of times that j changes when i=1 and the number 1000 of times that j changes when i=1000)

Since quantifiers and set formers are in effect pre-packaged iterations, very similar rules apply to them. To evaluate an existential quantifier like

```
(2)     ...EXISTS x IN s|C(x)...
```

will take a time to equal to the number of items x examined multiplied by the average time required to evaluate the Boolean condition C(x). If the quantifier (2) evaluates to FALSE, then all the members of s will need to be examined, so the time required will be s multiplied by the average time to evaluate C(x). If (2) evaluates to TRUE, then iteration over s will terminate as soon as an x satisfying C(x) is found; since iteration over a set is performed in a somewhat unpredictable order, the number of iterations needed to find such an x should be roughly #s/(#sat+1), were -sat- is the

set of all x satisfying the condition C(x).

Similar remarks apply to setformers and tupleformers, except that

(a) each insertion into a set takes somewhat longer than a simple iterative step, because of the necessity to check for and eliminate duplicate elements, and

(b) The implicit iteration appearing in a set or tuple former like

```
{x IN s|C(x)}
```

must always proceed until all the elements of s have been examined.

As an application of these rules, note that execution of the harmless looking code fragment

(3)
```
f:={ };

(FOR x IN s)
  f(x):={y IN s|(EXISTS z in s|C(x,y,z))};
END FOR;
```

involves three nested loops: first the FOR-loop which appears explicitly, next the implicit iteration over s in the setformer {y IN s..}, and then finally the implicit iteration over s in the quantifier EXISTS z IN s... Therefore the number of cycles required to execute (3) can be as high as the cube of the number of elements of the set s.

The possibility that a program can loop forever in an ill-constructed WHILE loop should serve to alert us to the fact that analysis of the time required to execute a WHILE loop can be much subtler than FOR-loop analysis. Of course, some WHILE loops are easily analysed. For example, if the variable k is not modified in its body, the loop

```
k:=0;

(WHILE (k+:=1)<n AND t(k)/=OM)
   ...
END WHILE;
```

behaves in very much the same way as a FOR-loop, and therefore will terminate after no more than n-1 iterations. On the other hand, consider the loop

(4)
```
t:=n*[0];

(WHILE EXISTS x=t(i)|x=0)
   print(t);
   t(i):=1;
   t(1..i-1):= (i-1)*[0];
END WHILE;
```

This begins by generating a tuple t:=[0,0,..0] of n zeroes, and then repeatedly sets the first nonzero coordinate of t to 1 and all the coordinates preceding this coordinate to zero, thus carrying out a (left-to-right) form of binary counting. The sequence of tuples printed is

```
[0,0,0,...0]
[1,0,0,...0]
[0,1,0,...0]
[1,1,0,...0]
[0,0,1,...0]
[1,0,1,...0],  etc.
```

and is plainly of length 2**n. Hence the number of cycles required to execute the WHILE-loop (4) is at least 2**n, which means that if n=50 the loop will execute for roughly 320 years even if 100,000 iterations are performed per second.

For a more realistic example of the way in which WHILE loops are typically used, consider the bubble sort

```
(5)     (WHILE EXISTS i IN [1.. t-1]|t(i)>t(i+1))
           [t(i),t(i+1)]:=[t(i+1),t(i)];
        END WHILE;
```

This searches a tuple t for out-of order components and interchanges a pair of such components whenever one is found. Plainly the number of cycles required to execute (5) is the average time required to search the tuple t for an out-of-order pair of adjacent components, multiplied by the number of interchanges required to put t in sorted order. Even though precise analysis of these times requires a close and subtle analysis going far beyond the scope of this book, it is not hard to estimate these time requirements crudely. We can guess that, as long as an out of order pair exists, one such pair will be found after searching through some fraction of the length of tuple t being sorted; thus evaluation of the existential quantifier appearing in the first line of (5) is estimated to require c* t cycles, where t is some constant which we will not attempt to evaluate here. Moreover, since each execution of the body of the WHILE-loop (5) corrects exactly one case in which a pair of elements appears in inverted order, the expected number of times that (5) must iterate to put t into its final sorted order should be roughly equal to the number of pairs of components of t which occur in inverted order. In a random arrangement of the components of t, roughly half the components to the left of a given t(i) should be larger than t(i), and roughly half the components to the right of t(i) should be smaller than t(i). Thus each component t(i) of t should appear in roughly #t/2 inverted pairs, and it follows, since t has #t components (and since this way of looking at things counts inverted pars twice, once for each of the components in such a pair) that the expected number of inverted pairs in a randomly chosen arrangement of the components of t should be roughly (1/4)*((#t)**2). Multiplying this expression by c*#t, representing the estimated time required to evaluate the existential quantifier in the first line of (5), we arrive at

```
(6)     (1/4*c)*((#t)**3)
```

for the time required to execute the bubble-sort code (5).

The approximations which we have made in arriving at the estimate (6) are too crude for the constant (1/4*c) appearing in (6) to be a good estimate. (Exercise 7.6.2 outlines an experimental procedure for estimating this coefficient more accurately.) The significant feature of the estimate (6) is that it tells us that the time required to sort a tuple by the bubble-sorting method is proportional to the cube of the length of t, i.e. that sorting a tuple of length 10 by this method should take roughly 120 cycles, sorting a tuple of length 100 roughly 120,000 cycles, and sorting a tuple of length 1000 roughly 120,000,000 cycles. These figures, which are not very favorable, reflect the rapidity with which the cube of n grows as n increases; in the jargon of algorithm analysis, one says that bubble-sort is 'an n cubed algorithm'. Clearly, any sorting algorithm whose time requirement grows less slowly than the cube of the length of t will be very much superior to bubble sort as a technique for sorting large tuples t.

7.5.2 Efficiency analysis of recursive routines

That the behavior of recursive routines resembles that of WHILE loops has already been pointed out in Section XXX. Like WHILE loops, recursive procedures can fail to terminate properly, and this warns us that even if they terminate they can execute for a very long time, so that careful analysis is needed to estimate their efficiency. On the other hand, recursive procedures can sometimes be surprisingly efficient. To show this, we will analyse the performance of just one rather simple recursive procedure, namely Floyd's quicksort, which has already been presented in Section 4.4.1. This procedure, which can sort any homogeneous set s of integers, real numbers, or strings, is simply

```
(7)     PROC quick_sort(s);
        IF  s=1 THEN RETURN [ ];END;
        x:=ARB s;
        RETURN quick_sort ({y IN s|y<x})
               +[x] + quick_sort ({y IN s|y>x});
        END PROC quick_sort;
```

Let T(n) be the number of cycles that this procedure will typically require to sort a set of n elements. Counting the set s can require a number of cycles proportional to n, and building up the two sets which appear in the final RETURN statement of (7) will require a like number of steps. Thus the time required to execute (7) is equal to some small constant multiple c*n of the length n of t (c=3 is a fair guess), plus the time required to execute the two recursive invocations of quicksort which appear in the second RETURN statement of (7). Since typically the element x chosen from s by the ARB function of (7) will lie roughly halfway between the largest and the smallest elements of s, each of the two sets {y IN s|y<x} and {y IN s|y>x} should contain approximately half the elements of s. Thus, given that T(n) is the time required to sort a collection of n elements by the quicksort method, sorting each of these sets by use of quicksort should require roughly T(n/2) cycles. It follows that T(n) satisfies the recursive relationship

```
(8)     T(n)=2*T(n/2)+c*n.
```

The first of the terms on the right of (8) represents the time typically required to execute the two recursive invocations of quicksort appearing in (7), and the term c*n represents all the work needed to prepare for then two invocations.

Having now derived the relationship (8), it is easy to solve it, and thus to arrive at an explicit estimate for T(n). To solve (8), we substitute the expression (8) for the occurence of T on the right of (8), getting

```
(8A)    T(n)=c*n+2*c*(n/2)+4*T(n/4)
            =2*c*n+4*T(n/4),
```

and then substituting (8) for T(n) on the right of (8A) we get

```
(8B)    T(n)=2*c*n+4*c*(n/4)+8*T(n/8)
            =3*c*n+8*T(n/8).
```

Continuing inductively in this way we will clearly get

```
T(n)=4*c*n+16*T(n/16),
T(n)=5*c*n+32*T(n/32),
```

and so forth, until eventually, when the power of 2 in the denominator on the right becomes roughly equal to n (which will happen after log n steps, where log n designates the logarithm of n to the base 2), we will find that T(n) is roughly

```
c*n*logn+n*T(1),
```

i.e., that T(n) can be estimated as the product of a small constant c (still roughly 3), times n, times the logarithm of n. One therefore says, in the jargon of algorithm analysis, that quicksort is an 'n log n' algorithm.

For n at all large and c roughly equal to 3, c*n*log n will be vastly smaller than the cube of n. For example, for n=1000, n**3 is 1,000,000,000, whereas c*n*log n is only 30,000. Therefore quicksort can be used to sort large tuples, which could not be sorted in any reasonable amount of time using bubble sort. For example, if #t=10,000 and on a computer capable of executing 100,000 of our nominal instruction cycles per second, sorting t using the bubble sort method will require approximately 16 hours, whereas quicksort will accomplish the same operation in roughly 4 seconds.

This simple example shows why it is so important to find algorithms whose time requirements do not rise rapidly as the arguments passed to them grow larger. Very considerable efforts have been devoted to the search for such high efficiency algorithms during the past decade, and a great many ingenious and important algorithms having this properly have been devised. Unfortunately, most of these algorithms lie beyond the scope of the present introductory work. For basic accounts of this important material, see the Bibliography which follows Chapter XI.

7.5.3 More about the efficiency of the primitive SETL operations. A warning concerning value copying.

Some SETL operations, like s WITH x, where s is a set and t WITH x where t is a tuple, also s LESS x, x FROM s, x FROMB t, and x FROME t, modify a composite object (i.e. a set s or tuple t) which may be large. The same remark applies to the tuple assignment t(i):=x, and to map assignments like f(i):=x and f{i}:=x. The time required to execute these operations will vary dramatically, depending on whether or not the large composite argument of the operation needs to be copied.

To understand this important point, note first of all that copying is sometimes necessary. Consider, for example, the code

```
(1)     s:={1,2,3,4,5,6,7,8,9,10,15,20};
        s1:=s;
        s WITH:=25;
        s1 LESS:=2;
        print('s=',s,'s1=',s1);
```

The output that this will produce is

```
s= {1 2 3 4 5 6 7 8 9 10 15 20 25} s1= {1 3 4 5 6 7 8 9 10 15 20}
```

Since two different values will have been created by the time we reach the final step of (1), it is clear that somewhere along the way to this final step the single set constant assigned to the variable s will have to be copied. This (logically necessary) copying can be done when the value of s is assigned to the second variable s1 in the second line of (1) (copying on assignment), or can be done in the third line of (1), when the value of s (but not that of s1) is modified by addition of the extra element 25 (copying on modification). Where copying actually takes place will depend on the particular version of the SETL compiler that you are using, and especially on whether or not this compiler includes an 'optimisation' phase. But in any case, some copying is necessary, and copying a set or tuple with n components always requires n cycles. Hence execution of an apparently harmless operation like t(i):=x can require a number of cycles proportional to the length of t.

On the other hand, copying is frequently unnecessary, and both the optimising version of the SETL compiler and the SETL execution-time support system include mechanisms for avoiding copying when it is not logically necessary. (Since these are implementation-level mechanisms, and fairly complex ones at that, we shall say little about how this is done.) When no copying is involved, the operation s WITH x is only two or so times slower than the membership test x IN s, and similar remarks apply to the other operations in the group we have been considering. For example, the assignment t(i):=x can be done in just one of our nominal 'cycles', and in the same circumstances map assignment is roughly five times as slow.

Equality and inequality comparisons between composite objects are also interesting operations to consider. To perform an equality (or inequality) comparison between two tuples, we first compare their lengths. If these are the same, the tuples may be equal, but even so their equality must be

checked by iterating over them in parallel and checking corresponding components for equality. A similar technique is used to determine equality of two sets when both of these have equally many members: we iterate over one of the sets, verifying that each of its members is also a member of the other set. Hence if they are different, comparison of two sets or tuples for equality may require only a few cycles, but if they are equal a time proportional to the size or the sets or tuples being compared may be needed to compare them.

Declarations which instruct the SETL compiler to modify the data structures which it uses to represent SETL objects will be described in Chapter X. These declarations allow set operations like 'x IN s', 'y=f(x)', 'f(x):=y', etc., to be handled as efficiently as tuple operations like 'y:=t(x)' and 't(x):=y' Used properly, they can increase the efficiency of set operations like s1+s2. s1*s2, and s1=s2 very substantially.

7.5.4 Data structures for high-efficiency realization of important operations

We have seen in the preceding pages that execution of some of the most important operations which SETL provides, for example set union and tuple concatenation, requires a time proportional to the size of the arguments to which these operations are applied. However, if the programs performing these unions and concatenations use the relevant sets and tuples only in restricted, particularly favorable ways, we can sometimes improve their efficiency very greatly by alternate, more complex representations for the objects appearing in these programs. Although doing this is something of a violation of the 'SETL spirit', which emphasises ease of programming over efficiency of program, SETL can be used to explain these efficiency-oriented techniques. Of course, it is important to understand these techniques because efficiency sometimes becomes an essential issue.

To focus our discussion of this matter, let us consider the problem of concatenating two tuples t1 and t2. SETL permits this concatenation to be formed simply by writing t1 + t2, but, as indicated earlier, execution of this operation can involve copying all the components of t1 and t2, and require a time proportional to the sum of the lengths of t1 and t2. It is therefore of interest that an alternative 'list' representation of tuples can sometimes be used to produce this concatenation much more rapidly.

In this representation, a tuple t:=[x1,...xn] is represented using a pair of mappings, called -next- and -val_of-, and by a variable -compl- which locates the first component x1 of the tuple. The first component of the tuple is val_of(compl), the next component is val_of(next(compl)), etc. Thus the -next- mapping steps us along from one component 'index' to the next 'index' (we will use atoms for these 'indices'), whereas the -val_of- mapping gives us the actual component value associated with each 'index'. The last 'index' cn in the list is distinguished by having next(cn)=OM.

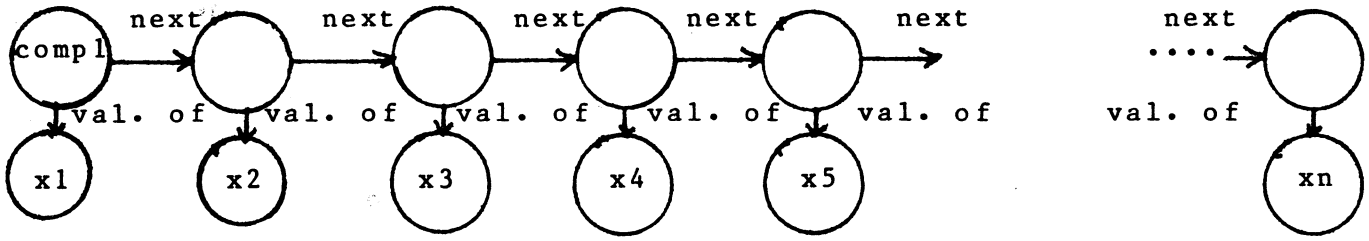


Figure 7.1. A tuple represented by a 'chained list' of elements.

At first sight, representing a tuple in this way may not appear to be a very good idea. Of course, it is not hard to iterate over tuples having this representation: we simply start at -compl- and apply -next- repeatedly to step along, always applying -val_of- to get the component value corresponding to whatver index we have reached. For example, instead of writing

```
(1A)     IF EXISTS x = t(i)|C(x) THEN...ELSE...
```

as we would if t were a standard SETL tuple, we would write

```
(1B)     i:=compl;  $ initialise search index
         (WHILE i /= OM)
           x:=val_of(i);
           IF c(x) THEN QUIT WHILE; END;
           i:=next(i);  $ advance search index
         END WHILE;
         IF i/=OM THEN ... ELSE ...
```

Although no less efficient than (1A), the code (1B) is certainly more complex and harder to read than (1A). Moreover, finding a given component t(k) of t is much less efficient in the list representation, since for standard tuples the operation

```
x:=t(k)
```

is performed in one or two cycles, whereas if t has the list representation we will instead have to execute the code

```
i:=compl;

(FOR j IN [1..k-1]) i:=next(i); END;

x:=val_of(i);
```

whose execution requires at least k cycles. On the other hand, other important tuple operations can be performed much more rapidly in the list representation than for standard SETL tuples. For example, for a standard tuple the operation which inserts x immediately after the i-th component of t requires time proportional to the length of t, since to create the expanded tuple all of the elements of t will have to be copied into new positions. On the other hand, if t has the list representation, this operation can be done in just a few cycles, since all we have to do is

(a) create a new atom ix to serve as the 'index' for the new component value x;

(b) link ix at the appropriate position into to list representing t.

Similar remarks apply to the operation t(i..i):=[ ] which deletes a given component from a tuple in list representation. The following two procedures represent these operations. In writing these procedures, we suppose that a single pair of maps -next- and -val_of- will be used to represent all tuples, and that the variables -next- and -val_of- have been declared global. We also suppose that only one logical reference to any of the tuples we consider is ever extant, so that no copying (see the preceding section) ever needs to be performed.

```
PROC insert(x,i):
$ inserts x immediately after the tuple component whose index is i

next(ix:=NEWAT):=next(i); $create a new index ix, and make it
```

```
                              $the predecessor of next(i)
    next(i):=ix;              $and the successor of i
    val_of(ix):=x;

    END PROC insert;

    PROC delete(i);
    $delete the component immediately following that whose index is i

    next(i):=next(next(i)?i); $ unless i is the last index in its tuple
                              $ make i's successor the successor of i's
                              $ original successor
    END PROC delete;
```

Provided that neither t1 nor t2 will be required after t1 and t2 are concatenated, and that the index i of the last component of t1 is easily available, the concatenation of t1 and t2 can also be formed in a number of cycles independent of the length of t1 and t2. The following procedure, in which we assume that each tuple t in list form is represented by a pair [first,last] consisting of the first and the last index of t, shows this:

```
    PROC concat(t1,t2);

    [t1_first,t1_last]:=t1;  $ 'unpack' the first and last
                             $                          indices of t1
    [t2_first,t2_last]:=t2;  $ and the first and last indices of t2

    IF t1_first=OM THEN       $ t1 is an empty tuple
        RETURN t2;
    ELSEIF t2_first=OM THEN   $ t2 is an empty tuple
        RETURN t1;
    ELSE
        next(t1_last):=t2_first; $ link the two lists
        RETURN [t1_first,t2_last];
    END IF;

    END PROC concat;
```

Quite a few other trick representations of tuples, sets, maps, etc. are known. If the family of operations applied to a SETL object is appropriately limited, use of one of these special representations can be very advantageous. Since further exploration of this very important issue would take us beyond the scope of the present work, we refer the reader to the bibliography appearing after Chapter XI for additional material concerning the issue of 'data structuring'.

7.6 Exercises

Ex. 1 How could you use the techniques described in Section XXX to make nonterminating recursions less likely to occur?

Ex. 2 Take the Bubble-sort procedure described in Section 4.1.1 and the Merge-sort procedure described in Section 4.4.2, and modify them by inserting code which will count the number of comparisons which they make when used to sort a given vector t. Use these modified routines to sort tuples of length 50, 100, and 200, counting the number of comparisons performed, and measuring their relative efficiencies. Try tuples with random components, and also try tuples with only a few components out of sorted order.

Ex. 3 The following SETL code is syntactically correct, but very poorly laid out. Put it in a better format, and add appropriate comments.

```
PROGRAM sort;read(s);t:=[ ]; (WHILE s/={ })s LESS:=(x:=MIN/s);
t:=[x]+t;END;print(t);END;
```

Improve the following program, which is also correct but poorly laid out.

```
PROGRAM find_palindromes;  $'Madam Im Adam'
LOOP DO read(x); IF x/=OM THEN y:=+/[c:c IN x|c/=' '];
IF y=[y(j):j IN [#y,#y-1..1]]THEN print(x);END;ELSE
QUIT;END;END;
```

Ex. 4 What cases should be run to test the following recursive sort procedure comprehensively?

```
PROC sort(s); $ recursive sort of a set of integers

RETURN IF (x:=ARB s)=OM THEN [ ]
    ELSE sort({y IN s|y<x}) + sort({y IN s|y>=x}) END;

END PROC sort;
```

Run your tests, and try to estimate how thoroughly they test this code.

Ex. 5 Suppose that G is a graph represented as a set of ordered pairs. If G contains a cycle C then C can be regarded as a subset of G such that for each x in G there exists a y in C (namely the edge y folowing the edge x in the cycle C) such that x(2)=y(1). Conversely, if this condition is satisfied, then there exists a cycle, since starting with any x in C we can find an x' such that x(2) = x'(1), then an x'' in in G such that x'(2)=x''(1), and so on, until eventually we must return to an edge that occurs earlier in the chain x,x',x'',..., at which point a cycle will have been formed. This leads to the following program for testing a graph to see if it has a cycle:

```
PROGRAM test_for_cycle, $ tests any graph for the existence
                          $ of a cycle
read(G);
print(IF EXISTS C in POW(G)|
    FORALL x IN c|EXISTS y IN c|x(2)=y(1)
```

```
            THEN 'There exists a cycle'
            ELSE 'There exists no cycle' END);
    END PROGRAM test_for_cycle;
```

Work out a good battery of tests for this program, test it, and try to estimate how comprehensive your tests really are.

Ex. 6 In the quick_sort program shown in Section X, change the expression

```
sort({y IN s: y<x}) + [x] + sort({y IN s:y>x})
```

to

```
sort({y IN x:y<x}) + sort ({y IN s: y>x}),
```

thereby introducing a bug. Then run the erroneous program. What hapens? Could you guess the problem from the symptom? What would be a good way of debugging this program?

Ex. 7 Suppose that the subexpression [x] in Exercise 6 is not omitted, but is accidentally mistyped as [z]. What will happen? Why? What will happen if it is accidentally mistyped as [0]? As {x}?

Ex. 8 For debugging purposes, it is useful to have a monadic operator .OUT such that .OUT x always has the value x, but such that 'evaluation' of .OUT x prints the value of x. A binary operator s .OUT2 x which return x but prints both s and x can also be useful. Write definitions for these operators. How might you use them to debug the faulty recursive procedure described in Exercise 6?

Ex. 9 Each string in the following set consists of characters which are easily mistyped for each other:

```
{'1I|/', '7>', 'L>', 'D0O0', 'S5s', 'Z7', 'UVuv', '6b',
    '4+t*', '_-', 'GC6'}.
```

Write an expression that converts this set of strings into a set s consisting of all pairs of letters that are easily confused for one another. Use this set to create a 'bugging program' B, which can read the text of any SETL program P, introduce one randomly selected character substitution chosen from P into it, and write the erroneous version of P thereby produced into a file. Collect various short sample programs from your friends, 'bug' them using B, and ask your friends to see if they can spot the error. Then debug these programs, to see how long it takes you to track down the errors which B has introduced. If B is modified so that it never changes characters in SETL keywords, but only in identifiers, how much more elusive do the bugs that it introduces become?

Ex. 10 Suppose that the statement

```
[t(i), t(i+1)]:=[t(i+1),t(i)];
```

in the bubble-sort procedure of Section 4.1.1 is replaced by

```
(*)    t(i):=t(i+1); t(i+1):=t(i);
```

What would happen? If we checked the resulting version of bubble_sort by

adding

```
ASSERT FORALL i in [1.. t-1] | t(i) <= t(i+1);
```

would the problem introduced by the change (*) be found? What checking assertion could we write to catch the sort of error that (*) introduces?

Ex. 11 Suppose that in the bubble-sort procedure of Section 4.1.1 we inadvertently wrote [1..#t] instead of [1..#t-1]. What would happen? If we wrote [#t..1] instead? If we wrote [1..#t-2]? If we wrote [2..#t]?

Ex. 12 Take the bubble-sort procedure shown in Section 4.1.1 and find at least three errors that might plausibly be made in writing or typing it, any of which would cause the code to loop endlessly. None of these errors should involve changing more than six characters. Take these erroneous versions of bubblesort to friends, and see how long it takes them to spot the errors. (This problems assumes that at least three of your friends know how to program!)

Ex. 13 Take the merge sort program shown in Section 4.4.2. Then modify it, to produce four different erroneous versions of merge sort, each of which contains one of the following list of common bugs. (Try to make your modifications as plausible, and as hard to spot, as possible.)

(i) Boolean condition stated in reversed form.
(ii) one branch of an IF statement omitted.
(iii) Premature exit from a loop.
(iv) Input data not checked, data of the wrong type read.

For each of these erroneous versions of estimate the time that would be required to find the error if you did not know where it was. Write a battery of tests sufficient to show that there is something wrong with each of these erroneous programs.

Ex. 14 Repeat exercise 13, but for the buckets-and-well program shown in Section 4.3.1. Produce five erroneous versions of this program, each with one or two plausible errors in every procedure. Devise a debugging plan which could discover most of these errors quickly. In what order does it seem best to test the procedures of this program? Where would it be most useful to place ASSERT statements? Try to devise assertions that can be checked quickly, but whose verification will be string evidence that the program is working as it should.

Ex. 15 The following version of quicksort contains just one error. What is it?

```
PROC quicksort(t);  $ t is assumed to be a tuple of integers

IF t=[ ] THEN RETURN; END;

x:=t(1);
t1:=[y:y=t(i)|y<x];
t2:=[y:y=t(i)|y=x];
t3:=[y:y=t(i)|y>x];
quicksort(t1); quicksort(t3);
```

```
    t:=t1 + t2 + t3;

    END PROC quicksort;
```

Ex. 16 How many of the errors by the 'bugging program' described in Exercise 9 could be found more easily using a program which reads SETL programs and prints out a list of all identifiers appearing only once in them?

Ex. 17 Write a maintainance test which could be used to check a sort program by comparing its results with those of quicksort. Use this test to verify that the merge-sort procedure shown in Section 4.4.2 is correct.

Ex. 18 Write a SETL system maintainance test which computes fifty set- or tuple-related values in two radically different ways and compares the results obtained. Your test should exploit various set-theoretic identities. For example

```
    {e(x): x IN DOMAIN f} = {e(x): [x,y] IN f}
                          = {e(x): y=f(x)}
```

should be true for every map f, and

```
    s1*s2 = s1-(s1-s2)
```

should be true for every pair of sets.

Ex. 19 To see what parts of a program have been executed in a series of tests, we can inroduce a global variable called POINTS, and a macro

```
    MACRO POINT(k); POINTS LESS:=k ENDM;
```

Then if we initialise POINTS by writing POINTS:={1..n}, insert a sequence of statements POINT(j), j=1,...,n into the code being tested, and print POINTS at the end of execution, each remaining member of POINTS will represent a section of code that has never been executed.

Apply this technique to develop a comprehensive set of tests for the bank accounting program described in Section 5.4.3. Add tests to your set until the condition POINTS={ } is achieved, to make sure that your collection of tests does not leave any section of code unexecuted.

Ex. 20 A boundary test for a program P is a systematic collection of tests which exercises P in all the legal but extreme cases which P is suppose to handle. Work up several such boundary tests for the bank accounting program described in Section 5.4.3. Your tests should include items like checks for $0.00, empty transaction files, etc.

Ex. 21 The bank accounting program described in Section 5.4.3 is totally unprotected against bad input. Modify it so that all input is systematically examined for acceptability; your input-examination procedures should check for all remotely plausible input errors. Write an English-language explanation of the input errors for which you check.

Ex. 22 Take one of your programs, approximately 10 lines long. Strip all

comments from it, and then introduce one misprint, to cause a bug (not one that syntax analysis would find.) Give the result to a friend (a good friend!) with a 3-line explanation of what the program is supposed to do. See if your friend can find and fix the error without expending more than an hour's effort.

Ex. 23 Develop test data for the GCD program outlined in Exercise XXX. Your tests should include cases in which the data is zero, negative, etc. and should test all relevant combinations of 'extreme' data of this kind.'

Ex. 24 Write the 'MISSING_SECTIONS' procedure described in Section XXX.

7.7 Formal Verification of Programs

The growing importance of programs to banks, airlines, engineering firms, insurance companies, universities, indeed to all the major institutions of our society, lends an inescapable importance to the question of program correctness. Once a program has been written, how can we be sure that it is correct, i.e. that when given legal input it will always produce the output that its author desires? This is a deep question, whose systematic exploration would take us far beyond the boundaries of the present introductory text. Nevertheless, in order to shed some light on the issues involved, we will use the present section to say something about it.

To begin with, we emphasise that mere program testing, even systematic testing like that described in Section 7.4, can never prove a program's correctness in any rigorous sense. Testing, to repeat an important maxim of the Dutch computer scientist Edsger Dijkstra, can show the presence, but not the absence, of bugs. Though systematic testing is an essential tool of program development, in asserting the rigorous correctness of a program we are asserting that it will run correctly in each of a potentialy infinite family of cases. Clearly, no finite sequence of test cases can cover all of them, and so any rigorous assertion that a program functions correctly in all possible cases must rely on some sort of mathematical proof.

The basic raw material out of which such proofs can be built is not too hard to find. When a programmer has written a program and checked it carefully for the first time, why does he believe that it will run correctly? If legitimate, this feeling of correctness must always rest on a comprehensive logical analysis of the conditions that arise as control moves from point to point during the execution of a program.

To show what such analysis involves, we will take a very simple program, namely one which calculates the product of two integers n and m by adding n to itself in m times. (The basic technique that we will use to prove the correctness of this trivial program is entirely general; however, the mass of technical detail needed to handle more complex examples grows rapidly, and to avoid this it is best to stick to a rudimentary example.) Since it is a bit easier to handle WHILE loops than FOR loops, we write our multiplication code as follows:

```
(1)     prod := 0 ;
        iterations := 0;

        (WHILE iterations /= m)
          prod := prod + n ;
          iterations := iterations + 1 ;
        END WHILE;
```

To begin to prove this program correct, we must first supplement it by adding a formal statement of what it is that the program is supposed to achieve. This can be done by adding an ASSERT statement at the very end of the program, giving us

```
(2)     Line1:  prod := 0 ;
        Line2:  iterations := 0 ;

        Line3:     (WHILE iterations /= m)
        Line4:         prod := prod + n ;
        Line5:         iterations := iterations + 1 ;
        Line6:     END WHILE ;

        Line7:  ASSERT  prod = m*n;
```

In (2), all lines of the program have been labeled to facilitate later reference. Note that addition of the final ASSERT statement is an absolutely necessary preliminary to any attempt to prove anything at all about the program until we have stated what a program is supposed to do, we cannot even begin to prove that it does what it is supposed to! This is to say that all rigorous proofs of program correctness are really proofs that two different descriptions of a computation, one a deliberately very high level, mathematical statement (like the final line in (2)) of what an algorithm accomplishes, the other a more detailed procedure (like the rest of the code (2)), really say the same thing.

This fundamental principle being understood, we go on to remark that in proving a program correct what one basically needs to do is just to write down the logical relationships between data items upon which the programer's understanding of his program's behavior rests. However, these relationships must be written down in a sufficiently complete manner, and must be expressed formally, using additional ASSERT statements.

To see what is involved, let us first analyse program (2) informally. If the author of (2) wished to convince a skeptical colleague that it really does compute the product m*n, what facts about (2) would he point out, what more detailed analysis would he offer? The crucial fact upon which program (2) depends is that each time the loop starting at Line 3 begins to repeat, the variable -prod- will be equal to the product of the variable -iterations- by the variable -m-. This is certainly true on the first iteration, since then both -prod- and -iterations- are zero, so we certainly have

(3) prod=iterations*n

(i.e. 0=0*n) on first entry to the loop. But if (3) is true at the start of k-th iteration, it must also be true at the end of the k-th iteration, since the body of the loop increments -prod- by n and -iterations- by 1. Hence (3) remains true during every iteration. But since the loop only terminates when the variables -iterations- and -m- are equal, (3) implies that prod=m*n at the end of the loop, which is what we wanted to prove.

The argument we have just presented is a satisfactory informal proof of the correctness of the program (2). Nevertheless, it is not quite what we require. In proving that a program is correct, we aim to rigorously exclude the possibility of any small, easily overlooked program 'bug'. For this, merely informal, English-language proof is insufficient, since such proofs are no less likely than programs to contain small errors. Moreover, some of the likeliest errors in programs (for example, counting in a manner that is off by 1) correspond closely to errors that occur frequently in mathematical

proofs (for example, starting a mathematical induction at the wrong place or missing one among multiple cases that a proof needs to examine.) Therefore, when we set out to prove a program rigorously correct, we must aim at something more formal and machine-checkable than an ordinary English-language proof of the kind ordinarily found in textbooks.

7.7.1 Formal Verification using Floyd assertions: general approach

This observation drives us to a more formal approach, like that devised by Robert Floyd, for proving programs like (2) correct. Floyd's formalism requires us to add ASSERT statements to a program P that we are trying to prove correct. These auxiliary ASSERT statements, sometimes called the 'Floyd assertions' for P, must satisfy two principal conditions:

(a) Enough ASSERT statements must be added so that there can exist no indefinetely long path through the program P which does not pass through at least one ASSERT statement. Another way of putting this is to say that at least one auxilary ASSERT statement must be inserted into every loop in the program P.

(b) Consider any one of these auxiliary ASSERT statements. It will have the form

(4) ASSERT C

where C which can be any Boolean-valued expression, is called the condition of the assert statement. The auxiliary assertion (4) will occur at some specific place in the progrm P, say, to be specific, immediately after Linej of P. Then we require C to assert every fact about the state of the program's variables that is relevant at Linej, i.e. every fact upon which the correct functioning of P from Linej onwards will depend. This important rule ensures that all the essential facts needed for proving the corectness of P are explicitly and formally written down in the auxiliary assertions added to P, and this is what makes a rigorous proof of correctness possible in principle.

Once the required assertions (4) have been added to P, we proceed as follows. Starting either at the first statement of P or at some one of the auxiliary ASSERT statements in it, we move forward line-by-line through the program along every possible path (i.e. path of control flow, which is to say path that the program could follow during its execution. All possible paths through P which start at an ASSERT statement but do not pass through any ASSERT statement must be considered one after another.) Because (by condition (a) above) there are no infinite loops not passing through an ASSERT statement, there will exist only finitely many such paths, and each such path will be bounded in length.

Tracing out all such paths q, we will use each of them in the following way to generate a set V of verification clauses. (With one exception, noted in (f) below, the verification clauses associated with a particular path q collect logical relationships between variable values which are certain to hold along q.)

(a) Suppose that the path q starts at an ASSERT statement ASSERT C, where C is a Boolean formula. Then we begin by putting C into V as its first clause. (This simply reflects the fact that C is assumed to be true at the

start of q.)

(b) If the path q passes through an assignment statement of the form

(A) x := expn;

(where expn can be any expression) we introduce a new variable identifier x′ (this identifier simply designates the value which x has after execution the assignment (A)) and add the clause

(B) x′=expn

to V. Occurences of x encountered later along the path q (but prior to any subsequent assignment to the same variable x) are then replaced by occurences of x′. (But at and after any later assignment to x we replace x by yet another new identifier x′′.) For example, the sequence of assignments

x := x+1; y:=y+1; z:=x+y; x:=x+z;

would generate the clauses

x′=x+1, y′=y+1, z′=x′+y′, x′′=x′+z′.

These rules just reflect the fact that the new value x′ which the variable x takes on immediately after the assignment (A) satisfies the equation (B), and that x retains this value until it becomes the target of a subsequent assignment.

(c) If the path q passes through an assignment of the special form

(C) x:=ARB s;

where s is some set-valued expression, then just as in paragraph (b) above we introduce a new name for x, but in this case we add the clause

(D) x′ IN s

to V. (This reflects the fact that the ARB operator can select an arbitrary element of s, so that (D) asserts everything we can know about the new value x′ given to the variable x by the assignment (C).)

(d) Conditional and unconditional GOTOs: If the path q passes through a control statement of the form

(E) GOTO Label;

then the path q must continue with the statement following the Label that appears in (E), but we add no clause to V at this point, since a simple GOTO does not test any condition or change the value of any variable.

On the other hand, if the path q passes through a control statement of the form

(F) IF C THEN GOTO Label; END;

then the path q can go on either to the statement immediately following (F) or to the statement following the Label that appears in (F). In the first case, we add the clause NOT C to V, in the second case we add the clause C to V. These rules simply reflect the fact that NOT C must hold if and when q passes through a control statement (F) without the instruction GOTO Label applying, but that C must hold if and when q reaches (F) and the instruction GOTO Label is applied.

(e) The rules for more complex control structures, for example general IF-constructs, WHILE loops, and UNTIL loops, can be deduced by rewriting them in terms of the more primitive constructs (E) and (F) and then applying the rules stated above. For example, if q encounters a multi-branch IF statement of the form

```
(G)   IF C1 THEN
        block1
      ELSEIF C2 THEN
        block2
      ...
      END IF;
```

and then enters block2, it is obvious that we must add the two clauses

```
NOT C1, C2
```

to V. Later, if and when q passes from the last statement of block2 to the first statement following the multi-branch IF, no clause needs to be added to V, since this transition, like (E), counts as an unconditional transfer.

THe rules applying to a WHILE loop

```
(H)   (WHILE C)
          body
      END WHILE;
```

are similar. If and when q passes through the WHILE header, either by entering the loop from the statement immediately prior to (H) or by 'looping back' from the final statement of the body of (H), we must add the Boolean clause C to V. On the other hand, if the path q encounters the WHILE header, but then leaves the loop (H) immediately, we must add the negated clause NOT C to V.

When q encounters the END WHILE line in (H) it will go immediately to the loop header standing at the start of (H). Since this is an unconditional transfer we add no clause to V in the case.

When q enters an UNTIL loop we need not add any clause to V since entry to such a loop is unconditional. However if and when q encounters the END UNTIL terminating such a loop the action that we must take is a bit more complex. Suppose, to be specific, that the loop in question has the form

```
(I)   (UNTIL C)
          body
      END UNTIL;
```

If, after encountering the END UNTIL statement, q exits the loop, then plainly we must add the clause C to V. On the other hand, if q encounters the END UNTIL clause and then loops back and continues with the first statement of the body of the loop, we must add the negated clause NOT C TO V.

(f) Eventually, the path q that we are following will end at an ASSERT statement

ASSERT C′

Our aim is then to show that C′ is necessarily true at the end of q, provided that the assertion C at which q starts (see (a) above) is true at the beginning of q, and provided also that program execution does indeed follow the sequence of steps corresponding to q. It is most convenient for this purpose to add the negated condition

NOT C′

to V. After doing this our aim must be to show that the set V of clauses is inconsistent, i.e., that not all the clauses of V can be true simultaneously. This is equivalent to requiring that, taken all together, the clauses of V, other than its final clause NOT C′, imply the condition C′.

(g) To complete the set of rules stated in the preceding paragraphs, we would need rules that tell us how to handle PROCEDURE definitions and invocations. However, since these rules are somewhat more complex than those stated above, we omit them. This means that the rules stated suffice for the formal verification of programs containing no procedure invocations, but not for programs which make use of procedures. This deficiency is not serious - it would not be terribly hard to remedy it - but to do so would take us beyond the limits proper to the present introductory work.

Once we have taken a program P containing ASSERT statements and generated the set V of verification clauses corresponding to each path q starting and ending at an ASSERT statement (but not passing through any other ASSERT statement), we are in position to prove the correctness of P mathematically. To do this, we must prove mathematically that each of the clause sets V which we have generated (i.e. each of the clause sets corresponding to a path q) is inconsistent. Suppose that we can succeed in doing this. We can then note that the clause CI initially placed in V is true by assumption, and that, with the exception of the final clause CF of V (see (f) above) all the other clauses inserted into V are true in virtue of the very meaning of the statements which the path q traverses. Hence, by showing that V is inconsistent, we will have shown that if CI is true at the start of q, then CF is true at the end of q. Once this has been demonstrated for every path q through the program P (or, more precisely, every path which connects two ASSERT statements but does not through any other ASSERT statement), it will follow by mathematical induction that every ASSERT statement written into P must evaluate to TRUE whenever it is reached, provided only that the ASSERT statement standing at the very head of P is true at the moment that execution of P begins. (This initial ASSERT statement, often called the input assertion of P, will normally summarise all the assumptions concerning input data on which the program P relies.)

All in all, we will have shown that the truth of every assertion written into P follows from the assumption that its input assertion is true.

It is important to realise that this final step of a formal program verification, i.e. the step of proving that each set V of verification clauses corresponding to a path q between ASSERT statements is inconsistent, is a purely mathematical task. I.e., when we begin this task we will already have decoupled the work which remains from any entanglement with the control structures and other programming dictions present in the original program P. It is precisely in order to achieve this, i.e. precisely in order to transform our original program-related verification task into a purely mathematical question, that we go to the trouble of reducing the program P to the collection of clause sets V that it generates. Note again that, once all the necessary Floyd assertions have been written into the text of P, generation of the clause sets V using the rules stated above is a simple mechanical matter, essentially a matter of systematic variable renaming and extraction of suitable portions of the statements encountered along each of the paths q.

7.7.2 Formal verification using Floyd assertions. An Example.

To apply the formal verification technique just outlined to the example (2) considered above, we must insert an auxiliary ASSERT statement into the WHILE loop appearing in the example. We choose to insert this ASSERT statement immediately after Line3 of (2). Call this place p. As explained, this added assertion must put on record every condition C which always holds at p and which would appear, implicitly or explicitly, in an informal proof of the correctness of the program (2). Since we have already given an informal proof that this simple program is correct, we already know what the inserted statement should say (namely it should say that (3) is always true at the beginning of an iteration.) Such an assertion is easily written and inserted; doing so, we obtain

```
(5)     Line1:  prod := 0
        Line2:  iterations := 0;

        Line3:  (WHILE iterations /= m)
                ASSERT prod=iterations*n;
        Line4:  prod := prod + n;
        Line5:  iterations := iterations + 1;
        Line6:  END WHILE;

        Line7:  ASSERT prod = m*n;
```

Writing (5) puts us in position to generate the clause sets needed to verify the correctness of the program we are considering. There are just four paths through this program that need to be taken into account. The first of these is the path running from the start of (5) to the first ASSERT statementin (5). By the rules stated above, this path generates the clause set

```
(6)     prod'=0, iterations'=0, iterations' /= m,
              NOT (prod'=iterations'*n)
```

The second path that we need to consider runs from the start of (5), to the WHILE-loop header but not into the WHILE-loop, and then to the final ASSERT statement. This path generates the clause set

```
(7)     prod'=0, iterations'=0, NOT (iterations'/=m),
            NOT (prod'=m*n), TRUE.
```

(Note that the final TRUE clause in both (6) and (7) can as well be dropped, since such an assertion, being logically vacuous, can never contribute to a logical contradiction.)

A third path between ASSERTs runs from the ASSERT statement following Line3, through the body of the WHILE loop, and then back to this same ASSERT statement. This generates the clause set

```
(8)     prod = iterations*n, prod'=prod+n, iterations'=iterations+1,
            iterations' /= m, NOT (prod'=iterations'*n).
```

The fourth and final path that we need to consider is the one which runs from the ASSERT statement following Line3 through the body of the WHILE Loop, but then exits the loop, passing through Line3 and then going immediately to Line7. The rules stated above tell us that this path generates the clause set

```
(9)     prod=iterations*n, prod'=prod+n, iterations'=iterations+1,
        NOT(iterations' /= m), NOT (prod' = m*n)
```

These are all possible paths not running through any ASSERT statement, and hence are all the clause sets that we need to consider. Once these clause sets have been generated it is easy to prove that each of them is inconsistent. In view of the simplicity of our original example, nothing more than elementary algebra is needed for any of these proofs. In (6), the first two clauses plainly contradict the fourth clause; in (7), the first three clauses contradict the fourth. In (8), the first three clauses contradict the fifth; in (9), the first four clauses contradict the fifth. This completes our formal verification of the program (2).

It is important to note that this formal verification is very close in spirit to the informal, English-language proof of the correctness of (2) that we gave earlier; the formal proof only regularises and systematises the informal proof. However, this formalisation has the vital effect of making it possible to proceed mechanically, thereby ruling out the possibility of small errors. Strictly speaking, for error to be impossible,, the clause sets would have to be generated mechnically by an extension of the SETL compiler, and the informal proof of inconsistency which we have supplied for to each clause set would have to be checked mechanically. This can be done, but not easily. As already observed, the clause-set generation process that we have applied to the example program (2) is quite general, and will apply with much the same ease to any other long or short program which has been decorated with a sufficiently full set of assertions. However, for more complex programs the clause sets generated will not be as simple as (6), (7), (8), and (9). Program (2) involves algebraic operations only, and this is why the clause sets generated from it consist entirely of elementary algebraic formulae. Less elementary programs

generally involve both algebraic and set-theoretic operations, and this will cause set-theoretic expressions to appear in the Floyd assertions and hence in the clause sets associated with these programs. (Several programs illustrating this remark appear in the verification-oriented exercises of Section 7.9.) To show that such clause sets are inconsistent is considerably less trivial than to deal with the clause sets arising in the highly simplified example that we have considered. Nevertheless, with care and sufficient effort the proofs required to show clause set inconsistency can always be checked formally after they have been constructed, by using only the tools which formal mathematics and symbolic logic make available. In this sense, the formal ASSERT-statement based verification approach that we have described reduces the problem of rigorous program verifiction to a purely mathematical question, namely that of proving the inconsistency certain clause-sets written in a formal mathematical notion. This is as far as we will carry our discussion of formal verification, since to discuss the mathematical problems that must then be faced would take us outside the scope proper to an introductory text.

7.8 Formative influences on program development

At this point in our text we have presented programs ranging from the simple to the complex, and have discussed both the pragmatic methods used to test programs systematically and the considerably more formal techniques that can be used to prove their correctness rigorously. The present section will discuss a deeper but more amorphous issue, specificaly we will try to give some account of the formative influences which shape programs and which determine the features that programs typically exhibit. By gaining some understanding of this fundamental question we can hope to put other important issues such as program design and program testing into a helpful broader perspective.

To understand what underlying forces shape the development of programs, it is well to observe that ingredients of two fundamental sorts enter into the composition of a program. Material of the first kind serves to define user desires and expectations concerning an intended application, for example the nature of expected input, and of outpput, including output text formats, graphic output, prompts and warnings issued by interactive systems, error diagnostics generated by compilers, etc. This material, which often constitutes the overwhelming bulk of a particular application-oriented program, is motivated by user-oriented considerations having an intrinsically nonmathematical character. Material of a second, much more highly algorithmic kind also apears in programs. This internal program material creates the toolbox of operations which is then used to achieve whatever external behavior is desired. Depending on the relative weight of program material belonging to these two categories (external and internal), a program can be called an 'externally motivated' or 'internally motivated' program, an 'application' or an 'algorithm'; one might even say a 'superficial' or a 'subtle' program.

Looking back over some of the programs presented in earlier chapters, it is easy to apply this distinction. For example, the shortest path code presented in Section XXX of Chapter III is an internally motivated algorithm (though not a very deep one). In contrast, the cross reference program presented in Section 5.2.1.2 of Chapter V has very little algorithmic content; most of its details relate to such external matters as the rules which distinguish words from punctuation marks in English text, and one whole subprocedure, namely PROC XXX of this program, is needed only because we want to print lists of line numbers in a neat, easy-to-read tabular arrangement. Other examples are the quicksort procedure of Section ZZZ and the mergesort procedure of Section 4.4.2, which is algorithms whose recusive structure gives them a certain depth in spite of their brevity; and the polynomial manipulation procedures of Section YYY, which are also algorithms, albeit rather easy ones since they are little more than transcriptions of the ordinary algebraic definitions of polynomial sum, difference, product, etc. On the other hand, the 'turtle language' interpreter presented in Section ZZZ is externally rather than internally determined: this code uses no nontrivial algorithm, but merely reflects the rules of the turtle language in an almost one-to-one manner. The 'buckets and well' program of Section 4.3.1 makes the distinction between internally and externally motivated code particularly clear, since one of its procedures, namely the crucial PROC find_path, is an internally motivated algorithm (very close in spirit to the path-finding PROC XXX of Section ZZZ), while all its other procedures are externally motivated, some of these

relating to such issues as the acquisition and checking of initial data, while others merely serve to represent the rules of the 'buckets' problem itself.

The basic concepts and notations of mathematics, which SETL makes available as tools of programming, serve very adequately to define the internally motivated, algorithmic parts of programs. We have already seen that SETL's set-theoretic features allow mathematical functions to be described either in a deliberately succinct, 'high' style which defines them very directly, or more procedurally by algorithms which compute these same functions, sometimes in surprising, clever, much more efficient ways. We have also noted that useful mathematical operations which are not directly provided by SETL can be built up by writing suitable families of procedures, and have emphasised (see our discussion of the family of polynomial-manipulating procedures developed in Section XXX) that such families should be written to hide the internal representational details of the mathematical objects they manipulate, allowing a user to think in terms of these objects (e.g. polynomials) rather than in more primitive set-theoretic terms. By using such approaches, by studying important algorithms carefully, and by consulting the rapidly growing technical literature of algorithms, which by now describes many useful, highly sophisticated algoritms, you will find that the purely algorithmic side of programming can be brought under a reasonable degree of control.

The externally motivated aspects of programs reflect a considerably more miscellaneous congeries of influences, for example the physical or administrative structure of real-world systems; the form and sequencing of expected input and desired output; the reactions, including prompts and warnings, expected from interactive systems; heuristic approaches used to manipulate physical or symbolic objects effectively, etc. How can we come to terms with such varied material?

There has developed a large, though largely administrative literature concerning the important problem of how to come to terms with external aspects of application design before the start of detailed programming. This is the so- called problem of requirements specification. Concerning the literature devoted to this problem, the astute observer C.J. Myers comments: 'Although no methodology exists for external design, a valuable principle to follow is the idea of conceptual integrity, [i.e.]... the harmony (or lack of harmony) among the external interfaces of the system... The easiest way not to achieve coneptual harmony is to attempt to produce an external design with too many people. The magic number seems to be about two. Depending on the size of the project, one or two people should have the responsibility for the external design. ... Who, then, should these select responsible people be? ... The process of external design has little or nothing to do with programming; it is more directly concerned with understanding the user's environment, problem, and needs, and the psychology of man-machine communications... Because of its increasing importance in software development, external design requires some type of specialist. The specialist must understand all the fields mentioned above, and should also have a familiarity with all phases of software design and testing to understand the effects of external design on these phases. Candidates that come to mind are systems analysts, behavioral psychologists, operations-research specialists, industrial engineers, and possibly computer scientists (providing their education includes these areas, which is rarely

the case).'

Though Myers' general remarks are helpful, it is still important to try to say something more about the organisation of externally motivated, applications-oriented programs.

One important possibility in this area is to develop special applications-oriented programming languages whose objects and operations define useful standard approaches to important application areas. Even if such languages remain unimplemented and are not available to be run on any compputer, their notations and general conceptual structure can serve as important tools of thought. In particlar, in developing an application it may be well to write out a first version of the application in a helpful, even if unimplemented, auxiliary language. This first version can then be translated into SETL by choosing SETL representations for all the kinds of objects appearing in the auxiliary language and writing SETL routines which implement its primitive operations. Used in this way, the auxiliary language serves to tell us what families of operations can work harmoniously together, and into what procedures a SETL application code can most usefully be organised. For this reason, comparative study of numerous disparate application-orieted languages, for example SNOBOL, APL, GPSS, APT, COBOL, etc., is recommended as an intellectual exercise for the would-be programmer.

Another useful suggestion, which plays a role in the design of appication-oriented programming languages, is to strive deliberately to use general mathematical operations rather than tailored special cases of them in developing prototype applications. Contrasting with this recommended practice, ordinary application-oriented code tends to mix internally and externally motivated program material inextricably, i.e. output details are allowed to control the choice of algorithms, and opportunities to generate output which an algorithm seems to afford are allowed to determine much of what the end-user sees. The result is often an inartistic package, which meets user requirements only minimally, and which is full of redundant, hard to maintain, and inefficient algorithmic fragments. By separating external application design from choice and elaboration of internal algorithms much more cleanly, it should be possible to treat these two problems separately, and thus to arrive at more satisfactory solutions of both of them.

A related suggestion is to use well-designed, relatively general-purpose application packages as building blocks for the construction of more complex applications. Consider, for example, the problem of designing an interactive system into which formatted commands will be entered to elicit system responses. As part of the design of such a system, command input conventions and command decomposition routines always need to be developed. It may be possible to handle this command input task by adapting a standard text editor very slightly. If this is done, the suitably modified editor will also serve to define and implement command facilities which can be as flexible and successful as the editor itself. This example illustrates the way in which well-designed, flexible application modules can be used, alongside of internally-oriented mathematical operations, as building blocks for more advanced applications. What is desirable is to familiarise yourself with a library of application-oriented modules which can be used somewhat as one uses a library of algorithms, but with the significant difference that they address more application- and user-oriented issues.

As we have said, much of the text of an applications-oriented program is nothing more than a restatement, in programming language terms, of external facts and rules pertaining to the intended application. Once one has found a way of representing these facts and rules in a form which is a succinct and clear as a well-conceived English language description of these same details would be, one has programmed these external aspects about as effectively as can be expected. Beyond this, the algorithmic content of a highly 'external' program will normally be small. However, the following elements will often play some role:

(a) A few genuine but generally rather elementary algorithms may be used. For example one may want to sort, perform a binary search, or put the data to be processed into some arrangement which makes it easy to locate significantly interelated groups of data items.

(b) To improve efficiency, one will often apply the process of 'formal differentiation' described in Section XXX to an application-oriented code. As explained in Section XXX, this is the technique of speeding up the calculation of a quantity E that will be required repeatedly by storing its value in a variable value_of_E, which must then be updated whenever any parameter on which E depends is changed. (Whenever this common technique is applied, it tends to complicate the application code, since it replaces a single, integral, often self-explanatory computation of E by multiple scattered, harder-to-fathom updates of value_of_E,) A related technique is to replace direct use of set-formers and tuple-formers by loops which build these same values. Sometimes this is done in order to combine several such loops, all of which iterate over the same set, into a single loop. For example, in application-oriented code (and even in hand- optimised algorithms) one is less apt to see

```
(1a)  num_rich_families := #{x IN families |
                              family_income(x) >= 100000};
      num_middle_families := #{x IN families |
        family_income(x) < 100000  AND  family_income(x) > 5000};
      num_poor_families := #{x IN families |
                              family_income(x) <= 5000};
```

than to see something like

```
(1B)     num_rich_families := num_middle_families :=
             num_poor_families := 0;

         (FOR x IN families)

            IF (income:=family_income(x))>=100000 THEN
               num_rich_families+:=1;
            ELSEIF income>5000 THEN
               num_middle_families+:=1;
            ELSE
               num_poor_families+:=1;
            END IF;

        END FOR;
```

The code (1B) arises from (1A) by expansion into loops of the three

set-formers appearing in (1A), followed by combination of the three resulting loops, and then by the application of a few other rather obvious optimising transformations. Note that (1B), although much more efficient and not much lengthier than (1A), is not quite as obvious a piece of code; certainly (1B) is less brutally direct than (1A).

Internally motivated code passages, which is to say significant algorithms, use a much wider range of tricks than ordinarily appear in more superficial, application-oriented, programs. (It is partly for this reason that it is well to separate internally determined from externally determined code sections: externally oriented code can often be ground out routinely once a good approach has been defined, whereas deeper, internally-oriented code needs to be approached much more cautiously, more 'by the book'.) Formal differentiation, as described above, plays a great role in the design of internally oriented algorithms.

Another important technique of algorithm design is exploitation of recursive mathematical relationships which express some desired function f of a composite object x in terms of values f(x1),...,f(xn) calculated for one or more smaller subparts xj of x. As noted in Section 4.4, relationships

f(x)=g(f(x1),..,f(xn))

of this recursive kind underly such high-efficiency algorithms as mergesort and quicksort.

Beyond these two most common techniques, the ongoing work of algorithm designers has already uncovered many sophisticated techniques which can be used to accomplish a great range of important tasks with remarkable efficiency. Some of these algorithms rest on quite subtle mathematical relationships, whose discussion goes beyond the scope of this book. However, your ability to devise truly effective approaches to programming problems will be strongly conditioned by your familiarity with the rich and growing literature of algorithms, and you are strongly advised to proceed with the study of this material as soon as you have mastered the more basic material contained in this book. A short list of useful collections of more advanced algorithms is found at the end of this chapter.

7.9 Exercises

Ex. 1 Into the bubble-sort code shown as (5) of Section 4.1.1, insert code which will count the number of iterations performed.
Then:

(a) Measure this number I of iterations for a randomly chosen tuples of varying lengths L, and calculate the ratio of I to L**3, to estimate the constant C that should appear in the formula I=C*L**3 projected in Section 7.5.1. Do the same for the quicksort method of section 4.4.1.

(b) How much more efficient than the bubble sort method (5) of Section X would you expect the quicksort method (YYY) to be, for sorting a tuple of 10 elements? For sorting a tuple of 100, or 1000 elements?

Ex. 2 Take the Bubble-sort procedure described in Section 4.1.1 and the

Merge-sort procedure described in Section YYY, and modify them by inserting code which will count the number of comparisons which they make when used to sort a given vector t. Use them to sort tuples of length 50, 100, and 200, counting the number of comparisons performed, and measuring their relative efficiencies. Try both tuples with random components, and tuples with only a few components out of sorted order.

Ex. 3 Use the technique described in Section 7.5.2 to estimate the time required to sort a vector of length t using the merge sort algorithm shown in Section YYY, and also the time required to search for a specified component in a sorted vector using the binary search algorithm given in Section ZZZ.

Ex. 4 What set will be printed by the following code?

```
n:=10;
s:={ };

(For i IN [1..n]) s WITH:=s; END;
```

If we changed the first statement to n := 1000, for roughly how long would you expect the resulting code to execute?

Ex. 5 Compare the time required to execute the following codes:

```
n:=500;
s:={ };

(FOR i IN [1..n]) s WITH:=2*i; END;
```

and

```
n:=500;
s:={ };

(FOR k IN [1..n]) s WITH:=2*i, t:=s; END;
```

What accounts for the difference?

Ex. 6 Write a program which will execute the ten elementary SETL operations which you consider most important, 1000 times each, and from this will estimate the time required to execute each such instruction. To eliminate the time required just to execute looping operations, your tests should compare loops like

```
(FOR i IN [1..n]) x:=y+z; END;
(FOR i IN [1..n]) x:=y+z; x:=y+z; END;
```

The time difference per iteration is then clearly the cost of executing the additional operation.

Ex. 7 Take the buckets-and-well program described in Section 4.3.1 and modify it by inserting code which will count the number of times that every one of its procedures is called and the number of times that every loop in it is executed. This information should be written to a tuple, and a

general purpose routine which prints this information in an attractive format should be designed and implemented.

Ex. 8 One reason why the Eulerian path program shown in Section 11.1 is not as efficient as a reprogrammed version of it could be is that to build up the final Eulerian path it makes repeated insertions into the middle of the path p being developed. As explained in Section XXX, each such insertion forces us to copy p if p is represented in the standard way as a SETL tuple. A better possibility is to represent p by a 'list' of the form described in Section YYY, i.e. by a map f which sends each point of p into the next point, so that if x0 is the first point of p, then p is [x0,f(x0),f(f(x0)),..]. Rewrite the Eulerian path program to represent p in this way. Try for an efficient variant, e.g. one which avoids unnecessary sarching through lists.

Ex. 9 A tuple t all of whose components are different from OM can be represented in the 'list' form described in Section XXX, i.e. by a pair [x1,f] such that x1 is the first component of t and f is a map which sends each component of t into the next component of t. Use an iteration macro to write short codes which convert a tuple t from its standard form to this list form and vice-versa.

Ex. 10 Rewrite program (2) of Section XXX by introducing labels and GOTOs in place of the WHILE loop appearing in this program. More precisely, the WHILE-loop header should be replaced by the following labeled statement:

```
    Label1:  If iterations /= m THEN GOTO Label2; END;
```

and the WHILE loop trailer END WHILE should be replaced by the sequence

```
    GOTO Label1;
    Label2:     $ the final ASSERT statement of (2) should
                $ follow this label
```

If we transform (2) in this way we can insert the auxiliary assertion

```
    ASSERT prod=iterations*n;
```

immediately after Label1. Make this assertion; then generate clause sets as in Section 7.7.1 and prove that the resulting variant of program (2) is correct. How does this proof compare in difficulty to the proof of correctness of program (2) given in Section 7.7.2?

Ex. 11 A set-theoretic iteration

```
(1)     (FOR x IN s)
```

can be rewritten as a WHILE loop in the following way: We introduce a new variable s' (representing the collection of elements of s that have not yet been iterated over.) Then the loop header (1) can be rewritten as a WHILE loop header in the following way:

```
(1')    s':=s;

        (WHILE s'/={ })
```

```
x := ARBs'; s' := s'-{x};
```

(The END FOR corresponding to (1) must be replaced by END WHILE.) Applying this technique, prove that if s is a set of integers then the program

```
(2)    count1 := 0; count2 := 0;
       (FOR x IN s)
         IF x>0 THEN count1 := count1+1; END;
         IF x<=0 THEN count2 := count2+1; END;
       END FOR;
```

gives the variables count1 and count2 final values satisfying the equations count1+count2=#s. You are required to work out a full set of Floyd assertions for the program, and to write out the clause sets generated by these Floyd assertions. A rigorous English-language proof that each of these clause sets is inconsistent should then be given.

Ex. 12 Assume that s1 and s2 are two sets. Proceeding as in Exercise 11, prove that the program

```
count:=0;
(FOR x IN s1)
  (FOR y IN s2)
    count := count+1;
  END FOR;
END FOR;
```

gives the variable -count- a final value equal to #s1*#s2.

Ex. 13 Take the merge-sort program of Section 4.4.2 and introduce as many hard to-find bugs into it as possible. Give the result to a friend, and see if he can find all the bugs, and what is the average time needed to find one bug ?

7.10 References to material on alternative data structures. References for additional material on algorithms.

Reingold, Nievergelt, and Deo: Combinational Algorithms - Theory and Practice (Prentice-Hall Publishers, 1977) is an intermediate-level work which presents many useful techniques for generating combinatorial objects, fast searching and sorting, and graph processing. It also discusses the mathematical techniques used to estimate algorithm efficiency, and can serve well as a guide to further reading in this important area.

The Design and Analysis of Computer Algorithms by A. Aho, J. Hopcroft, and J. Ullman (Addison-Wesley Publishers, 1975), which is more advanced, contains an excellent survey of many important algorithms, data-structuring techniques, and methods for determining the efficiency of algorithms. This useful book also describes various important techniques for proving upper bounds on the speed with which various quantities can be calculated. The first three volumes of Donald Knuth's famous Art of Computer Programming (Addison-Wesley Publishers, 1973) cover several important classes of algorithms (including basic combinatorial algorithms, polynomial manipulation, multiprecision arithmetic, calculation of random numbers, sorting, and searching) very comprehensively. Knuth gives many

detailed analyses of algorithm efficiency and is the basic reference for this topic. Borodin and Munro, Computational Complexity of Algebraic and Numeric Problems (American Elsevier Publishers, 1975) is a specialised work which presents many algorithms for high-efficiency processing of polynomials and for related algebraic and arithmetic processes.

Numerical algorithms, i.e.algorithms for carrying out numerical computations, including solution of linear and nonlinear equations, calculation of integrals, solution of differential equations, minimisation of functions of several variables etc. have a very extensive history, which reaches back to the nineteenth century and beyond. A first-class modern inroduction to this classical area of computational technique is found in Dahlquist, Bjorck, and Anderson Numerical Methods. (Prentice-Hall Publishers, 1974)

Methods for treating systems of linear equations and inequalities form the content of the area of algorithmics known as linear progrmming. For an account of this interesting and important subject, see D. Luenberger, Introduction to linear and nonlinear programming. (Addison-Wesley Publishers, 1973).

Many areas of algorithm design have developed very actively during the last few years. One of the most fascinating of these is computational geometry, the body of techniques used for the rapid calculation of solutions to geometric problems. For an introduction to recent work in this area, see M. Shamos, Computational Geometry Ph. d. Thesis, Yale University(1978).

$

CHAPTER 8

ADDITIONAL I/O AND ENVIRONMENTAL FUNCTIONS; BACKTRACKING

Chapter 8: Additional I/O and Environmental Functions; Backtracking

In this chapter, we cover various SETL capabilities that have been ignored in the preceeding, more elementary chapters. These include additional facilities for input/output, for sensing aspects of the environment in which a SETL program is running, and for passing strings or integers as parameters to SETL runs in a particularly convenient way. A full account of all the memory options and listing control commands which can be used to modify significant aspects of SETL compilation and execution is given. Finally, we give an account of of an interesting, somewhat unuaual type of control facility which SETL supports: backtracking, which makes an intriguing kind of non-deterministic programming availale.

Chapter Table of Contents:

8.1 Input-output facilities

While less developed than those of some other languages, the input-output facilities of SETL are adequate for most ordinary applications. Facilities for reading and writing simple string input, structured input representing SETL objects, and input/output using an internal 'binary' format which can be handled more efficiently than SETL's structured input are all supported. Note that relatively powerful string facilities available in SETL can also be used to format text that is to be printed.

The SETL I/O operations deal with files of two kinds:

(a) 'Text' (also called 'coded' files), which can be read, either as sequences of lines (which are read in as simple character strings, using 'GET', described below), or as structured encodings of SETL objects (possibly extending over multiple lines; these are read in using 'READ').

(b) 'Binary' files. These can only be written using PUTB and can only be read using GETB (see below). These files store SETL objects in their internal representation, And are read or written more efficiently than coded files.

All files are treated in strictly 'sequential' fashion by the SETL I/O primitives. That is, a file is regarded as a logical sequence (either of strings or of SETL Objects) from which input can only be read sequentially, starting with the first item in the file, and reading through the file to its last item, until end-of-file is eventually reached. Read operations are performed by READ, READA, GET, or GETB, see below. Output operations (i.e. PRINT, PRINTA, PUT, or PUTB) always add items to the end of a file, thereby making it longer. At each moment, a given file can only be used either for input or for output, not both, and must be used one of the two mutually exclusive modes (a) or (b), depending on whether the file contains binary or coded information.

The input-output operations which SETL supports are as follows:

(i) OPEN(file,mode). This opens the file specified by its first argument, thereby making the file available for other operations. Both arguments of the OPEN operation are strings. The forms acceptable for the first argument are machine dependent since they are identical with the forms of file names as defined by the execution environment. For example, on the DEC VAX running under the VMS/2.0 operating system, the following file parameters would all be acceptable:

```
OPEN('data.','CODED');                  $ simple file name
OPEN('test.dat','BINARY-IN');           $ qualified file name
OPEN('[dewar.doc]book.txt','CODED');    $ directory name followed
                                        $ by file name
```

The second argument of the OPEN function must be one of the following strings:

```
'BINARY'      (same as BINARY-IN)
'BINARY-IN'   opens file for  input by GETB
```

'BINARY-OUT'	opens file for output by PUTB
'CODED'	(same as CODED-IN)
'CODED-IN'	opens file for input by READ, READA, and GET
'CODED-OUT'	opens file for output by PRINT, PRINTA, and PUT
'PRINT'	opens file intended for printing the file is opened for output. Files opened in this manner will include special 'carriage control' characters; see below for details.
'TEXT'	(same as 'CODED-IN')
'TEXT-IN'	(same as 'CODED-IN')
'TEXT-OUT'	(same as 'CODED-OUT')

The OPEN primitive returns the value TRUE if the operation of opening the file succeeds, FALSE if this operation fails. Since the OPEN operation always involves communication with an underlying operating system, the meaning of 'success' and 'failure' is environment-dependent to a certain degree. Generally speaking, however, opening a file for input will succeed if a file having the name specified in the OPEN operation is available in the operating environment and has not already been opened; opening a file for output will succeed if the file has not already been opened. Opening an already open file causes an error.

(ii) CLOSE(file). This terminates input/output to a file established by a prior call to OPEN, and releases the file to the operating environment.

(iii) GET(file,lhs1,...,lhsk): This gets successive lines from the specified file, and assigns them (as strings) to lhs1,...,lhsk in turn. (Here and below, lhs1,...,lhsk must be either simple variables or expressions which can legally occur on the left-hand side of an assignment statement.) Lines read by GET should not ordinarily be enclosed in quote characters; if quote characters occur in such lines, they will be treated not as string delimiters but as parts of the string being read. For example, if the first two lines of a file 'xxx' are

```
THIS IS LINE 1
'THIS IS LINE 2'
```

then the effect of the GET statement

```
GET('xxx',lna,lnb);
```

is exactly the same as that of the pair of assignments

```
lna := 'THIS IS LINE 1';
lnb :='''THIS IS LINE 2''';
```

If GET encounters end of data on the file that it is reading, it (like READ, see Section XXX) behaves as if it had read an OM.

To GET input from the standard input file, the standard file name 'INPUT' should be used.

(iv) GETB(file,lhs1,...,lhsk). This reads successive SETL objects from the specified file, and assigns them to lhs1,...,lhsk in turn. (As in the case of GET, lhs1,...,lhsk must be expressions which could legally appear on the left-hand side of an assignment.) In this case, the file being read must be a SETL binary file, and must have been opened by the command OPEN(file,'BINARY-IN'). Note that a SETL binary file will almost always have been created using PUTB.

If GETB encounters end of data on the file that it is reading, it behaves as if it has read an OM.

(v) PRINT(expn1,...,expnk). This writes the values of expn1,...,expnk to the standard output file. See Section XXX above for details.

(vi) PRINTA(file,expn1,...,expnk). This is similar to PRINT, except that its first argument is the name of a file (of 'CODED' type) to which the output produced by this operation is written.

(vii) PUT(file,expn1,...,expnk). This writes text lines to the file specified by its first argument, which must be of 'CODED' type. The expressions expn1,...,expnk must evaluate to strings. Each such expression causes a single line to be sent to the specified file.

(viii) PUTB(file,expn1,...,expnk). This writes the values of expn1,...,expnk to the specified file, which must be a SETL binary file, and must have been opened by the command OPEN(file,'BINARY-OUT'). Here expn1,...expnk can be arbitrary SETL values.

Provided that they involve no atoms, values written by PUTB can always be read back in GETB. (The special rules which govern the handling of atoms by PUTB and GETB are explained below.) Note that the very desirable symmetrical relationship between PUTB and GETB that this rule reflects does not hold for PRINTA and READA, simply because strings written by PRINTA will not include the quote marks which READA requires. Hence, if you want to write SETL Objects to external media for temporary storage and then read them back you must do so using PUTB and GETB, rather than PRINTA and READA.

(ix) READ(lhs1,...,lhsk). This reads a sequence of SETL values from the standard input file. (As in the case of GET, lhs1,...,lhsk must be expressions which could legally appear on the left-hand side of an assignment.)

If READ encounters end of data, it behaves as if it had read an OM.

(x) READA(file,lhs1,...,lhsk). This is similar to READ, except that its first argument is the name of a file (of 'CODED' type) from which the input produced by the READA operation will be obtained.

(xi) EOF. This is a nulladic operation which yields TRUE if the most recent input operation executed (which will be either a READ, READA, GET, or GETB operation) reached the end of the file being read; otherwise EOF yields FALSE.

Since every input operation affects the value of EOF, it may become necessary in some programs to save EOF values by assigning them explicitly to auxiliary variables.

(xii) EJECT( ) or EJECT(file). This writes a page eject character to the specified file, or, if no file is specified, sends a page eject to the standard output file. The file to which an eject command is directed must either be the standard output file or must have been opened using the command OPEN(file,PRINT). Only files opened in this way can accept carriage-control characters like the 'eject' character.

(xii) TITLE( ) OR TITLE(str). These operations initiate and suspend generation of titles for the standard output file. TITLE(str) must have a string-valued argument. TITLE causes a page eject on the standard output file, and establishes its argument as the title string, which then appears at the head of all subsequent pages (until the title is changed later by another TITLE command.) Titled pages are numbered sequentially. TITLE with no argument disables generation of titles. (See Section XXX for a related titling facility.)

Note that if the PUT primitive is used with a file which was opened by an OPEN(file,'PRINT') command and which is intended for printing, the first character of each line of the file printed will be treated as a carriage control character rather than a as a normal print character. Characters treated in this way will not be printed, and their presence may cause unexpected page ejects or other undesirable effects. For this reason, the PUT primitive should not be used in place of PRINT or PRINTA except by programmers familiar with carriage-control conventions.

The PUTB primitive can be used to write atoms to a BINARY file. These atoms can be read back by GETB. Note however that if a file containing atoms is read in by a program that has just started to run, regeneration of atoms will restart at atom number 1, and hence some of the newly generated atoms may appear to be identical with old atoms obtained from a file via GETB. To avoid difficulties in this case, it may be necessary to use some annoying artifice, e.g. to begin by generating many 'throwaway' atoms, until the last atom present in the data structure read in by GETB has been bypassed.

The input-output facilities described above can be used to write output interactively to a terminal (and acquire input from a terminal.) See Section XXX for the conventions that apply in this case.

8.2 Backtracking

'Backtracking' or 'nondeterministic programming' is an ingenious technique useful for solving a very common and important type of search problem. Such problems can be regarded as logical or combinatorial 'mazes' which a program must explore in order to find a desired solution point. In favorable cases, one will be able to do this by devising an algorithm which proceeds in relatively direct fashion from an initial position to a solution, along a path involving little or no 'trial and

error'. However, some problems are too complex for such algorithms to be available, and it is for these problems that the method of backtracking is most useful. Characteristically, programs for solving these problems encounter situations in which a decision must be made as to which of several alternatives is to be explored next, but in which no clear grounds can be found for making one rather than another decision. A correct decision will lead on to a solution of the problem being explored, but an incorrect decision will wind up in a dead end, and the program will have to revert to the point at which it took its first wrong turning and try an alternative originally overlooked. Finding paths through mazes and solving geometric and spatial puzzles like the well-known 'instant insanity' puzzle are obvious examples of this kind of problem.

The backtracking primitives to be described in this section make it easy to program solutions to these problems. Just two primitives, whose power at first seems almost magical, are required. These two primitives, whose workings we will describe in this section, are called OK and FAIL respectively.

OK is a (parameterless) Boolean-valued function, but one which we can think of as having a very major additional effect. More specificaly, wherever OK is called, we at once 'split' our program into two copies of itself, identical except that OK yields the value TRUE in one of these copies, and FALSE in the other. After splitting, both these copies continue to execute independently and in parallel. If either of these copies subsequently encounters another OK, it will split yet again in the same way. If it subsequently encounters an occurence of our second backtracking primitive FAIL (which is simply a parameterless statement) it will immediately cease execution and disappear. The problem that our program is solving becomes solved as soon as one of the many copies into which the program has split reaches a solution.

The way in which we really implement this kind of 'splitting' will be described later in this chapter. For the moment, let us simply assume that such splitting is possible, and note how powerful and general its effects are. Suppose, for example, that a program needs to make a simple binary choice, say to perform one of two complex calculations, but that no algorithm for making this choice at that point is known. Then we can simply write

```
IF OK THEN x := f1(x,y,z); ELSE x := f2(u,w,v); END;
```

This creates two copies of our program, one of which executes the invocation of f1, the other one of f2. If one of these copies subsequently encounters the statement FAIL it will simply disappear. Hence (ignoring implementation difficulties) we concentrate our attention on that 'lucky' copy of the program which eventually finds the problem solution that we are looking for. From the point of view of this lucky copy, OK has acted as a magical 'oracle': when called it returned one of the possible values TRUE or FALSE; the value chosen was always such as to steer the program past any lurking occurence of FAIL.

Note that OK can be used to make any kind of choice, to make multiple choices, and to chose among multiple alternatives. For example, consider the following statement:

```
IF OK THEN RETURN e; END;
```

This splits our program into two, one of which immediately returns with the value e, while the other continues executing the function in which the IF statement appears. This shows how 'extreme' a choice OK can make.

To explore multiple choices, we can for example write

```
IF OK THEN
   IF OK THEN x := north(y); ELSE x := south(y); END;
ELSE
   IF OK THEN x := east(y); ELSE x := west(y); END;
END IF;
```

This creates four copies of an initial program, within each of which one of the four functions north, south, east, west, will be invoked.

To choose among still more highly multiple alternatives, we can even write

```
IF EXISTS x IN s | OK THEN
   RETURN x;
ELSE
   FAIL;
END IF;
```

where s is a set. In this case, the iterative search triggered by the EXISTS construct will iterate over all of the elements x of s in turn. For each such element, OK will be evaluated. This will cause a split into two program copies, in one of which x will be considered 'ok' and will be returned, while in the other copy x will have been rejected and the iteration (i.e. the iterative search triggered by the EXISTS construct) will continue on to the next element of s, again splitting, etc. This will create as many logical copies of the original program as s has elements, in each one of which one particular element x of s will have been selected and returned. (It will also generate a copy in which no x is accepted and the EXISTS primitive yields FALSE; but this copy immediately executes a FAIL and disappears.) This useful backtracking fragment can be embedded in a function:

```
PROC choose(s);                    $nondeterministic choice procedure

IF EXISTS x IN s | OK THEN
   RETURN x;
ELSE
   FAIL;
END IF;

END PROC;
```

The net effect of a call to choose(s) will be to split the program executing it into as many copies as there are elements in the set (or string, or triple) s; each element (or character, or component) of s is the value returned by -choose- in one of these copies.

8.2.1 Implementation of Backtracking

To actually implement the logical 'splitting' implied by the OK primitive, one can proceed as follows. Each time OK is evaluated, make a complete copy of the state of the program in which it occurs. This should record the value of all variables, including temporary variables, the sequence of procedure calls outstanding, the instruction currently being executed, etc. Call all this information an 'environment', and save it somewhere on a stack. Then give OK the value TRUE, and continue the current computation. If the current computation succeeds in finding the solution it wants and terminates normally, nothing more is necessary. If, on the other hand, it subsequently executes a FAIL and disappears, then retrieve the last environment saved, and restart the computation from the state recorded in this environment, but this time give the OK which it is just a process of evaluating the value FALSE. (Note that each environment saved contains all the information needed to restart a calculation from a prior point in its history, and that each of these restart points represents a calculation in the very act of evaluating the function OK). It is clear that this process of serial exploration will eventually either find the solution being sought, or will work through the history of all the split computations generated by successive evaluations of OK, to discover that all of them FAIL. In this latter case, an error exit is taken, and a diagnostic message is issued:

```
*** EXECUTED -FAIL- IN PRIMAL ENVIRONMENT
```

The preceding paragraphs describe something very close to the way in which SETL implements the backtracking primitives OK and FAIL. This implementation allows the semantics of these operations to be modified slightly, in part to improve their efficiency, in part to allow other, occasionally useful, slightly more complex effects to be obtained. First of all, rather than saving the values of all variables whenever OK is executed, the SETL system requires an indication from the user as to which variables should be restored to their previous values after a FAIL. When FAIL is executed, only the values of those variables declared by the user to be backtrack variables are restored. (Of course, the system itself will restore the stack, program counter, internal variables, etc.). The variables which are to be restored to their previous values after a FAIL are declared in the following example:

```
VAR x,y,z :  BACK;
```

In the presence of this (and only this) BACK declaration, the attribute -BACK- would be attached to the variables x, y and z, and no others, and only those will be saved and restored on OK/FAIL.

We will illustrate the use of OK, FAIL, and the BACK declaration by using them to solve a simple but very well-known combinatorial problem, the so-called 8 queens problem, which can be stated as follows.

On an 8 by 8 chess board, place 8 queens (i.e. pieces that move up, down and diagonally) in such a fashion that no two queens attack each other.

Note that there is no obvious non-backtracking approach to the problem. However, the backtracking primitives allow it to be solved easily.

We simply place queens successively on the board, in apropriate unattacked squares, until all have been placed. The OK primitive is used (as an oracle!) to ensure that we never make the mistake of placing a queen on a inappropriate square. If there were queens still to be placed but no unattacked squares left, we would have to FAIL, but we can take the complacent attitude that the values returned by OK will prevent this from ever happening. If for the moment we omit the necessary BACK declaration, and postpone the easy subfunction which tells us which squares are unattacked, SETL code for solution of the 8-queens problem can be written simply as:

```
used :={};                          $ the set of board squares which
                                    $ are occupied by a queen
(WHILE #used < 8)                   $ While not all queens have been
                                    $ placed

    possible := safe();             $  squares which are not under
                                    $ attack

    IF EXISTS square in possible | OK THEN

      used WITH:= square;           $ put queen on one more square

    ELSE                            $ All squares are under attack.
        FAIL;
    END IF;
END WHILE;

print_board;                            $ Display the solution.
```

In order to complete this program, we must

a) Decide on the variables which must be backtracked.

b) Choose a representation for the board, and specify the function -safe- and the output procedure -print_board-.

The variables which need to be backtracked (i.e. restored to their previous values after a FAIL) are those which will be used before being redefined following some OK, and which also might be modified after an OK. In the code shown above, both -used- and -possible- must be backtracked. The iteration variable -square- need not be saved, because

whenever we backtrack it is precisely in order to discard some previously chosen square. Thus , we only need the declaration:

```
VAR used, possible: BACK;
VAR board;
```

The representation of the board, and the nature of the procedures -safe- and -printboard- are independent of our backtracking schema. For completeness, here is a possible description of these items:

b1) The board is a set of positions, each position being represented by a pair of coordinates in the range [1..8]. (More economical representations suggest themselves, and you may want to invent some).

```
board := { [i,j] : i in [1..8], j in [1..8]};
```

b2) The function -safe- iterates over all board positions, and discards the ones which are under attack by queens placed in used squares.

```
PROC safe();
RETURN {square IN board | (NOT EXISTS queen IN used |
                              attacks(queen, square)) };
END PROC safe;
```

Finally, the predicate -attacks(p1,p2)- establishes whether board positions p1 and p2 are mutually threatening:

```
PROC attacks(p1, p2);
RETURN
    (p1(1) = p2(1))      OR                  $ p1 and p2 are on same row.
    (p1(2) = p2(2))      OR                  $ or on same column.
    ((p1(1)-p1(2)) = (p2(1)-p2(2))) OR       $ or same upwards diagonal.
    ((p1(1)+p1(2)) = (p2(1)+p2(2)));         $ or same downwards diagonal.
END PROC attacks;
```

The procedure -print_board- is left as an exercise to the reader.

8.2.2 Total failure, and the generation of all solutions to combinatorial problems.

In all our examples so far, we have assumed that the problem we are tackling actually has a solution. This may not always be the case. For example, how would the queens program behave if we specified a board size which was smaller than the number of queens to place? In such a case, the program would search through all possible positionings of the queens on the existing board, and fail on each of them. Eventually, a final failure would be executed, for which no backtracking alternatives exist (all positions having been tried). At this point, the SETL system, having run out of options, would terminate execution in the manner indicated above, i.e.:

```
*** EXECUTED -FAIL- IN PRIMAL ENVIRONMENT.
```

If we do not know a priori whether our problem has a solution or not, we may want to ensure that our backtracking program does not terminate abruptly upon terminal failure, but gives us some information as to the nature of the unsuccessful search (e.g. the number of tries) and perhaps awaits further input; in a word, we want the program to retain control. This can be achieved by inserting a top-level -OK- to which we will fall back in case a search fails completely. This correponds to the following general backtracking schema:

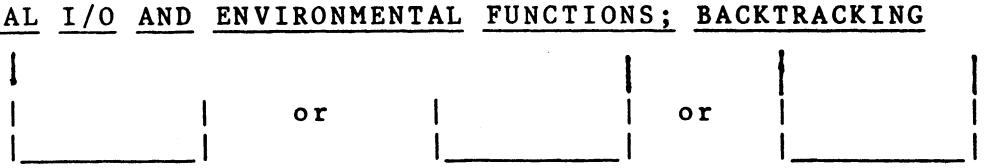
```
if OK THEN

    (while not complete(solution))

        possible_moves := moves(solution);

        IF EXISTS move IN possible_moves | OK THEN
            solution := update(solution, move);
        ELSE
            FAIL;
        END IF;

    END WHILE;

    display(solution);

ELSE

    print('Problem has no solution');
    ...                                  $ Actions upon find failure.
END IF;
```

As the example shows, information about the history of a backtracking computation can be gathered in non-backtracked variables, i.e. variables that do not appear in a -back- declaration. The values of non-backtracked variables are unaffected by the execution of OK and FAIL. An example of a variable that monitors the execution of a backtracking program is the variable -failure- in the tiling program shown below. The variable is used simply to count the number of times -FAIL- was executed.

8.2.3 Tiling problems.

The so-called 'tiling' problem can be stated as follows: given a set of square tiles of various sizes, find whether they can be used to cover a rectangular area of given height and length exactly.

To solve this problem by backtracking, we use the following approach: we keep track of the perimeter of the area which remains to be filled. Initially, this is just the perimeter of the rectangle to be tiled. At each step, the bottom of this perimeter must include a 'valley', i.e. a sequence of four vertices whose two middle ones are at a lower height than its first and last as shown in the following figure:



At each step of our exploration, we insert into the lower left corner of such a valley, one of the remaining tiles whose width is no greater than that of the valley, update our description of the perimeter of the area which remains to be tiled, and continue.

In the code that follows, the condition that determines the acceptability of a given tile is expressed as a conjunction: we want to find a tile among those remaining which fits(i.e. is no wider than) an existing valley, and which is OK, -(i.e. which will subsequently allow us to place all remaining tiles and complete the solution).

The only data structure of special interest in this program is the perimeter of the area remaining to be filled. It is described as a sequence of points, listed in counterclockwise order, starting from the upper left-hand corner of the area to be tiled. Thus, the original perimeter constitutes a valley, and the first tile to be placed goes in its lower left-hand corner. Each point on the perimeter is described by an ordered pair of coordinates. Further details of the algorithm can be gleaned from the commented code that follows.

```
PROGRAM tiling_puzzle;

$ This is a backtracking program that finds an arrangement of
$ given set of square tiles to fill in a specified rectangle.
$ The area still to be filled is specified the global variable
$ -perimeter-, which is the counterclockwise sequence of vertices
$ of the unfilled space that remains.
$ The algorithm proceeds by finding a valley in the bottom
$ of the empty area into which one the remaining tiles fits.

VAR perimeter,                    $ Of area to be tiled.
    placement,                    $ Of tiles already used.
    sizes_left,          $ Of available square tiles.
    count,                        $ Of tiles in each size.
    corner,                       $ Defining valley for next tile.
    next_size:                    $ To be tried.
                BACK;

VAR length, height,               $ Dimensions of rectangle to be filled.
    tiling;                       $ For display of successive placements.

$ The following macros establish some geometric vocabulary.

MACRO abcissa(i); perimeter(i)(1)                              ENDM;
MACRO ordinate(i); perimeter(i)(2)                             ENDM;

$ Macros describing properties of edges.

MACRO up(i);       (abcissa(i) = abcissa(i+1) AND
                    ordinate(i) < ordinate(i+1))               ENDM;
MACRO down(i);     (abcissa(i) = abcissa(i+1) AND
```

```
                    ordinate(i) > ordinate(i+1))          ENDM;

start:
print('enter length, height of area to be tiled');
read(length, height);
print('enter tuple of tiles to be used');
read(tiles);

$ Verify that tiles can cover exactly the specified area.

IF +/[t ** 2 : t IN tiles] /= length * height THEN
    print('no possible covering with this set');
    GOTO start;
END IF;

perimeter := [ [0,height], [0,0], [length,0], [length, height]];
placement := [];
sizes := {t : t IN tiles};
count := { [t, #[t1 : t1 IN tiles|t = t1]] : t IN sizes};
sizes_left := sizes;

failures := 0;      $ this variable keeps a count of the number of times
                                          we have backtracked

$ Define the topmost environment to which we will return in case of
$ complete failure.

IF OK THEN

  (WHILE sizes_left /= {} )                $ Continue placing tiles.

      $ Find valley in current perimeter: there must be one.

      ASSERT EXISTS corner IN [1..#perimeter-2] |
                 down(corner) AND up(corner+2);

    IF EXISTS next_size IN sizes_left | fits(corner,next_size) AND OK
       THEN

        count(next_size) -:= 1;

        IF count(next_size) = 0 THEN
            sizes_left LESS:= next_size;
        END IF;

        rebuild(corner, nextsize);         $ Fill in perimeter.
        printboard;                        $ Display solution so far.

        ELSE failures +:= 1; FAIL;

      END IF EXISTS;

  END WHILE;

    print; print('Solution:'); print; printboard;
```

```
ELSE
    print('no solution for this set' );
    print('backtracked ', failures,'  times');
END IF;


PROC fits(c,tile);

$ Determine whether -tile- fits in the valley defined by the points
$ c,c+1, c+2, c+3 in current perimeter.  Note also that we assuming
that all tiles are square.

RETURN
       (abcissa(c+2)-abcissa(c+1)) >= tile AND
               (height - ordinate(c+1)) >= tile;

END PROC fits;


PROC rebuild(i,tile);

$ A valley exists, delimited by points i to i+3, into which  -tile-
$ fits. Note the placement of the tile at the lower left corner (point
$ i+1 on the perimeter) and update the area which remains to be tiled.

placement WITH:= [perimeter(i+1), tile];

$ Calculate the position of the remaining vertices of the tile we
$ have just placed.

p1 := [abcissa(i+1), ordinate(i+1) + tile];
p2 := [abcissa(i+1) + tile, ordinate(i+1) + tile];
p3 := [abcissa(i+1) + tile, ordinate(i+1)];

$  new points on the perimeter of the (partially) filled valley.

new_points := [perimeter(i),p1,p2,p3,perimeter(i+2)];

$ discard edges of length zero.

IF p1 = perimeter(i) THEN new_points := new_points(3..); END IF;

$ eliminate hairpin turns.
IF p3(1) = abcissa(i+2) THEN
    redundant FROME new_points;
    redundant FROME new_points;
END IF;

$ Check for exact fit.
IF p2 = perimeter(i+3) THEN
    redundant FROME new_points;
    perimeter := perimeter(1..i-1) + new_points + perimeter(i+4..);

ELSE
```

```
    perimeter := perimeter(1..i-1) + new_points + perimeter(i+3..);
END IF;

RETURN;

END PROC;

PROC printboard;

$ Display succesive tiling arrangements.

$ As ordinarilily printed, the space occupied by a character is about
$ twice as tall as it is broad. To give our ouput the correct appearance
$ we double the number of character positions in the horizontal direc-
$ tion. Thus a square of size s whose lower-left corner is positioned
$ at  (x,y) is actually drawn with its vertices at character positions
$
$ [2*x+1, y+1], [2*x+s+1, y+1], [2*x+s+1, y+s+1], [2*x+1, y+s+1].
$
$ The area to be tiled is represented by a tuple of strings, one for
$ each horizontal line.


tiling := (height+1) * [ (2*length+1) * ' '];

(FORALL [[x,y], tile] IN placement )

    bottom := y + 1;
    top    := y + tile + 1;
    left   := 2 * x + 1;
    right  := 2 * (x + tile) + 1;

    $ draw top and  bottom of each square.
    tiling(bottom)(left+1..right-1) :=
        tiling(top)(left+1..right-1)    := (2 * tile -1) * '_';

    $ Complete upper corners of square, if they are not covered by
    $ another tile.
    IF tiling(top)(left)  = ' ' THEN
       tiling(top)(left) := '.';
    END IF;

    IF tiling(top)(right) = ' ' THEN
       tiling(top)(right):= '.';
    END IF;

    $ Draw sides of tile.
    (FORALL z IN [bottom..top-1])
        tiling(z)(left)  := '|';
        tiling(z)(right) := '|';
    END FORALL;

END FORALL;
```

```
$ Display tiling.

(FORALL i IN [#tiling,#tiling-1..1]) print(tiling(i)); END;

END PROC;

END PROGRAM tiling_puzzle;
```

8.2.4 Other uses of OK and FAIL

The -OK- and -FAIL- primitives are useful in other contexts than those of backtracking programs. We will now describe two such less obvious uses.

8.2.4.1 Combinatorial generators.

The generation of a set of combinatorial objects (all the subsets of a set, or all the permutations of a sequence, etc.) can often be given a simple description using OK and FAIL. We proceed the generation of each object from the desired set by an OK, and each time the construction of an object is completed, we execute a FAIL to force the generation of another object in the set. As an example of this, consider the problem of generating all the permutations of a set S. This can be done as follows: build a sequence by picking elements from S in any order; regard each choice of an element among the remaining ones as a backtracking point in order to force all possible choices to be made for a given position in the sequence. We also provide a top-level backtracking point, to which we return when all permutations have been generated. The following code shows the use of this technique:

```
PROC permutation_generator(S);

VAR S,x,perm : BACK;

perms := {};
perm  := [];

IF OK THEN                              $ Topmost backtracking point.

    (WHILE EXISTS x IN S | OK)
        S LESS := x;                    $ This element has been used.
        perm WITH:= x;                  $ And added to the current perm.
    END WHILE;

    IF S = {} THEN                      $ add to set of permutations.
        perms WITH:= perm;
    END IF;

    $ Now force a different choice.
    FAIL;
```

```
ELSE                                    $ All permutations have been
                                        $ generated.
    RETURN perms;

END IF;

END PROC permutation_generator;
```

8.2.4.2 Failures and exceptions.

The FAIL primitive can also be used to exit from a complex calculation in circumstances in which the calculation cannot proceed any further. This mechanism allows a form of error-handling which exists in some programming languages under various names (exceptions, ON-conditions, etc.) The need for such mechanisms is particularly clear when we consider recursive programs which may uncover an abnormal situation (for example invalid data) after a number of recursive calls. In such cases, it may become necessary to notify all pending recursive calls that an abnormal situation has arisen, and that the computation should not continue any further. This is a trifle awkward to program in the absence of some exception-handling mechanism. The OK/FAIL pair provides a simple mechanism of this type. We can establish a 'recovery point' at the top level of a program by writing:

```
IF OK THEN ....
```

The code attached to the ELSE branch of this conditional statement is executed when a FAIL is performed during program execution (assuming that this is the only OK in the program). This code functions as an 'exception handler' and the FAIL is said to 'raise the exception'. It is possible to program the handling of several exceptions, i.e. to execute FAIL under diverse abnormal circumstances, and note in some global variable (accessible to the exception handler) what the nature of the abnormal condition is.

8.2.5 Nondeterministic programs, or It Is OK After All.

There is another way of looking at the backtracking primitives just described, which adds nothing to the technical details of their workings, but sheds a different light on the meaning of backtracking. If we examine the sequence of choices made by a backtracking program which succeeds, then it is clear that those choices were correct: they led to the solution, after all! If we ignore the computer time which has been used, it is immaterial that the program may have come back several times to a certain OK, undoing its previous choice and trying something else; eventually, the proper choice was made. From the point of view of the end-result, each OK was infallible! We can therefore think of the OK primitive as an Oracle, which will somehow make the right choice when faced with various alternatives. (This explains the name: -OK- rather than a more tentative -TRY- for example). It is instructive to think of backtracking programs in this fashion, ignoring the trials and errors which will be executed by the running program, and instead seeing each OK as a point at which we have said (to the run-time

system): "You choose the right way. I don't know nor care how."

8.2.6 Auxiliary backtracking primitives.

SETL provides two additional primitives, SUCCEED and LEV, which allow additional control over the backtracking mechanism.

Housecleaning: the SUCCEED primitive.

Our description of the implementation of the OK primitive should make it clear that a price is paid for each execution of OK, namely storage must be used to preserve the value of backtracked variables and other run-time information. This information defines the environment in which an OK is executed. The storage utilized by each execution of an OK remains in use until execution of a subsequent FAIL brings us back to the environment in which that OK was executd. In the case of a program that reaches a solution (or partial solution) after executing a OK statement, the storage thus occupied is useless, because we will not fail again into that environment. If space starts to run short, we will want to release this reserved space. This can be accomplished by invoking the SUCCEED primitive. When invoked, all the information stored by the most recent execution of an OK, is erased from the system. Execution of a subsequent FAIL will no return us to the environment of that OK, but to some earlier one, if such exists. In other words, SUCCEED is a selective way of burning ones bridges behind one. Needless to say, this should only be done if the search has in fact succeeded.

8.2.6.2 Controlling the depth of the search: the LEV primitive.

The computational steps taken by a backtracking program can be seen to form a tree. Each node of this tree corresponds to some (partial) trial version of the solution being built. The descendants of a given node N correspond to the possible sequences of choices the OK primitive might make in moving forward from the situation corresponding to N. point in the computation. The root of the tree represents the starting state of the calculation. For example, in the 8 queens problem, the root of the tree corresponds to the empty board, the nodes immediately below this node correspond to possible placements of the first queen, etc. In the case of the 8-queens program, we can easily see that the full tree to be explored by the program has a height of 8 (counting the root to be at heigth zero) because there are only eight queens to be placed. For some backtracking problems, the height of the solution may not have an obvious upper bound, which means that the search may have to perform many tentative guesses (OKs) and may have to backtrack correspondingly many times. It is often necessary to know the current depth of the computation, i.e. the number of OKs which have been performed, and to which it may be necessary to backtrack on failure. The value of the system variable LEV is precisely that number. It is useful, when we happen to know that the solution for which we are searching cannot lie 'too deep' in this tree to cut off fruitless searches over unpromising parts of the tree. In such cases, we can, for example, write:

```
if LEV > maxlevel THEN FAIL; END;
```

8.3 Use of auxiliary 'Inclusion Libraries'

Carefully crafted procedures which perform common utility functions such as sorting, output formatting, parsing, etc. can be used over and over again in SETL programs. SETL supports several features intended to facilitate the use of such standard program libraries. One of these is the LIBRARY feature described in Section XXX, which makes it possible to bind pre-compiled collections of library programs into a composite program. (See Section YYY for additional material concerning binding of separately compiled programs.) In the present section, we will describe a simpler but related facility, which makes it possible to insert sections of source text gathered from an auxiliary 'inclusion library' into a SETL program that is about to be compiled.

An 'inclusion library' used in this way must be structured as a sequence of standard SETL source lines, divided into 'MEMBERS' by interspersed lines of the form

(1) .=MEMBER membername

Each such line introduces, and names, a 'member' of the inclusion library, which consists of all lines following the line (1), up to the next occurence of a line beginning with ' .=MEMBER'. (Note that the characters .=MEMBER in a header line like (1) must occupy character positions 2 thru 9 in the line; the first character in the header line must be blank.) The last member of the inclusion library extends from the header line which introduces it to the very end of the library.

All MEMBERs of an inclusion library must have distinct member names. To import a member -membername- of an inclusion library into a SETL program text P that is to be compiled, a line of the form

(2) .COPY membername

is required. This line must occur in P at the point at which the body of the inclusion library MEMBER introduced by line (1) is to be inserted. Like (1), the line (2) must begin in character position 2, following an initial blank. The -membername- in (2) must be identical with the -membername- in the line (1) introducing the text which is to replace (1).

During (the first, parse phase of) compilation, each .COPY line of the form (2) is replaced by the body of the MEMBER introduced by the corresponding line (1). For example, in the presence of an inclusion library containing the lines

```
...
 .=MEMBER constants
small_lets:='abcdefghijklmnopqrstuvwxyz'
big_lets:='ABCDEFGHIJKLMNOPQRSTUVWXYZ';
```

```
 .=MEMBER quicksort
PROC quicksort(s);
RETURN IF (x:=ARB s)=OM THEN [ ]
   ELSE quicksort({y IN s|y<x} + [x]
     + quicksort({y IN s|y>x}) END;
END PROC quicksort;

 .=MEMBER prettyprint
...
```

The source text

```
PROGRAM something;
 .COPY constants
PROC another;
...
END PROC another;
 .COPY quicksort
...
```

will be compiled exactly as if it read

```
PROGRAM something;
smalllets:='abcdefghijklmnopqrstuvwxyz';
biglets:='ABCDEFGHIJKLMNOPQRSTUVWXYZ';
...
PROC another;
...
END PROC another;
PROC quicksort(s);
RETURN IF (x:=ARB s)=OM THEN [ ]
 ELSE quicksort({y IN s|y<x}) + [x]
  +quicksort({y IN s|y>x})END;
END PROC quicksort;
...
```

The file used as inclusion library during a SETL compilation which makes use of the .COPY feature is specified by the control card parameter ILIB; see Section XXX and YYY(a) for additional details.

8.4 Listing-control commands

It is possible to alter the form of the listing which the SETL compiler produces by including listing control command lines of the form described below in your source program text. These commands, each of which must always occur on a separate line beginning with the two characters ' .', have no effect on compilation or execution other than to modify the form of the compilation listing. The allowed listing control commands are as follows;

```
NOLIST
 suspends listing of source text lines
LIST
 resumes listing of source text lines
EJECT
```

```
 advances compilation listing to new page
TITLE pagetitle
  This command specifies a pagetitle which will appear at the
   top of subsequent pages.
```

Note that the page title appearing in the preceding command cannot contain the apostrophe character, also that the AT control card parameter described in Section XXX can be used to request 'automatic titling'. If automatic titling is enabled, then each new PROCEDURE encountered will begin on a new page, which will be given a title derived from the PROCEDURE header line. (See Section XXX for the run-time equivalent of this compile-time command).

8.5 Environment operators and SETL command parameters.

SETL includes several facilities for sensing aspects of a program's external environment and for controlling optional details of compilation and execution. These facilities will be described in the present section.

(i) Parameterless keyword quantities.

The parameterless keyword

```
TIME
```

yields an integer representing the execution time, in milliseconds, used by your program from the start of execution up to the moment at which the TIME quantity is evaluated. This special quantity can be used to monitor the amount of processor time which your program is consuming.

The parameterless keyword quantity

```
DATE
```

yields a standard string consisting of the current day, date, and clock time, expressed as hours, minutes, and seconds. For example, the result of the command

```
print(DATE);
```

might be

```
SUN   01 MAR 81    14:49:13
```

(ii) Initial Program Parameters.

Integer or string parameters to be transmitted to a SETL program can be included in the operating system command-language line which initiates execution of the program. The precise external form in which these parameters should be given will depend to some extent on the operating system being used. For example, to transmit parameters P1 and

P2 with values 'YES' and 35 to a SETL program running under the DEC VAX VM 2.0 system, we would write

(1A) /P1=YES/P2=35

If running under the CDC CYBER NOS system we would have to write

(1B) (P1=YES,P2=35)

instead, and running under the IBM/370 CMS system we would write

(1C) (P1=YES P2=35)

(See Appendix XXX for an account of all the systems under which versions of SETL are available.)

Built-in functions called GETSPP and GETIPP are used to read these program command parameters. For example, to read the values of the string-valued parameter P1 appearing in the preceding examples and save the value in a variable x, we would write

```
x:=GETSPP('P1=defval/altval');
```

where -defval- and -altval- stand for arbitrary string constants. The GETSPP primitive searches the command line which initiated the SETL run for the occurence of a parameter definition of the form P1=abcde, where -abcde- can be an arbitrary string, or if the first occurence of P1 in the command line is not followed by an equal sign, simply for an occurence of the parameter name P1. Then

(i) If P1=abcde occurs on the command line, without a value being assigned to it, x is given the value abcde.

(ii) Otherwise, if P1 occurs on the command line, x is given the value -altval-.

(iii) Otherwise P1 is given the 'default' value -defval-.

The function GETIPP works in exactly the same way, except that it reads integer instead of string parameters, and supplies integer rather than string default values.

Suppose, for an example of all this, that the code

```
x1:=GETSPP('P1=LITTLE/BIG'); x2:=GETIPP('P2=1/0')
```

appears in a program being run under the DEC VAX VM/2.0 system. Then the appearance of the following parameter strings on the command line initiating a run of the program would give x1 and x2 the values indicated in the following table:

Command-Line Parameter String	x1 value	x2 value
/P1=MEDIUM/P2=2	'MEDIUM'	2
/P1=MEDIUM/P2	'MEDIUM'	0
/P1=MEDIUM	'MEDIUM'	1
/P1/P2=2	'BIG'	2
/P2=2	'LITTLE'	2
/P2	'LITTLE'	0
(no parameters)	'LITTLE'	1

A typical use of the GETIPP primitive is to switch on debugging or tracing facilities selectively. To do this, one can, for example, introduce a collection of variables called trace1, trace2,... etc. Debugging prints in your SETL program can then be made conditional on the values of these variables, e.g. by writing statements like

```
IF trace13=1 THEN
  print(...);   $ print appropriate debugging information
END IF;
```

If the variables trace1, trace2,...are initialised by statements

```
trace1:=GETIPP('TRACE1=0/1');
trace2:=GETIPP('TRACE2=0/1');
```

etc., then by passing TRACEj (i.e.,TRACEj=1) to a run (as a parameter of the command used to bring your SETL program into execution; see Section XXX) one can cause the corresponding trace output to be produced. Note that this will switch debug output on without you having to recompile the program being debugged.

Facilities very much like GETSPP and GETIPP are used in the SETL implementation, where they support the battery of compiler and run-time options described in the following sections.

8.5.1 Standard SETL command options

The SETL compiler and run-time systems themselves read a variety of control card parameters, using the GETIPP and GETSPP facilities of the SETL language. These parameters switch various compilation and debugging features on and off. In describing these parameters, we will use a notation typified by the example 'A=0/1', that is, the name of a parameter to be described will be given first, followed by an equal sign, followed by the 'default value' that the parameter will be given if its name does not appear in a parameter string followed by a slash, followed by the 'alternate value' that the parameter will be given if it is simply mentioned as a SETL command parameter name, but no value is explicitly assigned to it.

A parameter passed to the SETL system can be significant either to the parse, semantic analysis, or code generation phases of the SETL compiler, or to the SETL run-time support library, or to a SETL program containing an invocation of either GETSPP or GETIPP. The list of standard parameters which now follows lists parameters according to the

phase of the SETL system to which they are significant.

(a) Parse phase options.

AT=0/1 (automatic titling)

This option controls automatic titling of the parse phase output listing. AT=1 causes each SETL procedure to start a new page on the listing; AT=0 suppresses this automatic page advance.

CSET=EXT/POR (character set)

This option specifies the character set used in your SETL source. POR specifies that only the 'portable' subset of the collection of all possible characters is allowed; EXT specifies that both the 'portable' set and a wider class of 'extended' character sets are allowed. These character options allow or disallow the following character representations:

	'Portable' representation	'Extended' representation
left set bracket	<<	{
right set bracket	>>	}
left tuple bracket	(/	[
right tuple bracket	/)	]
'such that'	ST	| or !

(The printed characters shown above are intended to represent the corresponding ASCII standard internal codes. The actual characters printed or typed for these codes may vary from terminal to terminal, or from one printer to another.)

ETOKS=5/5 (error tokens)

The value of ETOKS controls the number of tokens listed in parse error diagnostic messages.

I=filename (input file)

The value of I specifies the name of the source file containing the SETL text to be compiled.

ILIB=filename (inclusion library)

As noted in Section XXX, text from an auxiliary 'inclusion library' can be imported into a SETL program being compiled. The value of ILIB defines the name of this inclusion library.

L=filename (listing file)

The value of L specifies the name of the standard 'listing' file to

which all compilation-phase output will be written.

LIST=0/1 (list compilation output)

The option LIST=1 causes a compilation-phase listing to be produced; L=0 suppresses this listing.

MLEN=1000/1000 (macro length)

The value of MLEN defines the maximum number of tokens allowed in a single macro body.

PEL=1000/1000 (parse error limit)

The value of PEL specifies the parse-phase error limit. If more than the specified number of errors are detected by the parse phase, compilation is terminated.

PFCC=1/0 (write printer carriage-control information)

PFCC=1 causes the output listing to contain carriage control information; PFCC=0 suppresses carriage control information.

PFLL=0/0 (line limit)

This command parameter is used in conjunction with PFPL; see the description of PFPL for additional information.

PFLP=60/0 (lines per page)

The value of PFLP determines the number of lines that will be printed on each output page.

PFPL=100/0 (page limit)

This parameter, together with PFLL, determines the amount of output that a program will be allowed to produce before being forcibly terminated. The limits imposed are as follows:

PFPL=0, PFLL=0
no output limit enforced.

PFPL=n, PFLL=0 (n>0)
a limit of n pages or n*PFLP output lines is imposed.

PFPL=0, PFLL=n (n>0)
a limit of n output lines is imposed.

PFPL=n, PFLL=m (n>0,m>0)
a limit of n pages or m output lines is imposed.

POL=filename ('Polish' file name)

This specifies the name of the 'parsed source' file passed by the SETL compiler's parse phase to its semantic analysis phase. See Section XXX for an explanation of the role played by this file.

TERM=filename (interactive terminal identification)

The SETL system will normally expect to write certain short messages, generally error and warning messages, to an interactive terminal. If no such terminal is available, or if for any other reason it is desired to write these messages to some other file, then TERM=filename can be used to designate this file. The option TERM=0 suppresses this 'terminal' output.

(b) Semantic analysis phase options

BIND=0/filename (binder file)

This parameter, and the associated parameter IBIND (see below) are used to pass seperately compiled files in Q1-format to the semantic analysis phase; see Section XXX for additional explanation. If either BIND or IBIND has a value different from zero, the semantic analysis phase will read various Q1-format files, and combine them with newly parsed SETL source input (named by the POL and XPOL parameters described below), producing a Q1-format file (named by the Q1 parameter described below). This output file represents the logical concatenation of all its input files in a parsed, semantically analysed form.

DITER=0/1 (modificatons during iteration are possible)

This option indicates whether the compiler can assume that objects being iterated over in a loop are not modified within the loop. DITER=0 disallows this assumption and causes the object being iterated over to be copied before an iteration begins; DITER=1 suppresses these copying operations.

IBIND=filename (auxiliary list of input Q1 files)

This parameter, and the associated parameter BIND (see above) are used to pass seperately compiled files in Q1-format to the semantic analysis phase. The file named by the IBIND parameter should itself be a list of file names, one name per record, these file names having whatever format is appropriate in view of the operating system under which the SETL compiler is running. All the files named in this list of files will be read and 'bound' together into the Q1-format output file which the semantic analysis phase produces. See parameter BIND (above), and also Section XXX for additional information.

L=filename (listing file)

Specifies the name of the standard 'listing' file to which all printable compilation-phase output will be written.

OPT=0/1 (optimisation)

Selecting the option OPT=1 causes a global optimisation phase to be executed between the normal semantic analysis and code generation phases. Note that this option has an effect only for implementatons which make the SETL optimiser available.

PFCC (carriage control)
PFLL (line limit)
PFLP (lines per page)
PFPL (page limit)

See subsection (a) above for details concerning these parameters.

POL=filename ('Polish' file name)

Specifies the name of the 'parsed source' or 'Polish' file passed from the SETL compiler's parse phase to the semantic analysis phase. See remarks concerning POL made in subsection (a) above.

Q1=filename ('Q1' file)

Specifies the name of the 'preliminary' code file passed from the SETL compiler's parse phase to its optimisation or code generation phase. Sec Section XXX for an explanation of the role played by this file.

SEL=1000/1000 (semantic error limit)

The value of SEL specifies the semantic analysis phase error limit. If more than the specified number of errors are detected during the semantic analysis phase, compilation is terminated.

SIF=0/1 (save intermediate files)

The option SIF=1 causes the 'preliminary code' or 'Q1' file produced by the semantic analysis phase to be saved. (See preceeding remarks concerning the parameter Q1. Normally this file will be deleted by the compiler's code generation phase.) Note that a file of this sort must be saved if SETL's separate compilation and 'binder' facilities are to be used; see Section XXX for additional details.

UV=0/1 (check for undeclared variables)

Selecting the option UV=1 will cause a warning message to be issued for each variable name used in your program which does not appear in any VAR statement. (This gives a handy way of ensuring that all variables appearing in the program have been documented, and for checking against accidental variable-name misspellings.)

(c) Code generation phase options

ASM=0/1 (produce assembler code)

The ASM=1 option will cause the SETL compiler to produce machine code for the computer on which you are running. ASM=0 will cause production of a less efficient but generally more compact interpretable code form. See Appendix XXX for more details concerning these options, which may not be implemented in all SETL systems.

BACK=0/1 (backtracking enabled)

The BACK=1 option allows generation of code supporting backtracking, and must be selected if the backtracking facilities of SETL (cf. Section XXX) are being used.

CA=0/0 (constants area size)

This code phase parameter is used to control the size of the 'constants area', which stores the values of constants appearing in your program. The option CA=0 sets the constants area size equal to half the initial memory size allocated for your program (see parameter H, below). If a positive value less than 1024 is specified for CA, then this value, multiplied by 1024, becomes the constants area size; thus CA=2 is equivalent to CA=2048. See Section XXX for additional information about the way in which the SETL system uses memory.

CEL=1000/1000 (code generation error limit)

This specifies the code-generation phase error limit. If more than the specified number of errors are detected by the code-generation phase, compilation is terminated.

H=0/0 (heap size)

The value of H specifies the initial virtual memory size that will be used when program execution begins. If H=0 is selected, an implementation-dependent default initial memory size is used. If a positive value less than 1024 is specified for H, this value, multiplied by 1024, becomes the initial memory size; thus H=2 is equivalent to H=2048. See Section XXX for additional information about the way in which the SETL run-time system uses memory.

L=filename (listing file)

Specifies the name of the standard 'listing' file to which all printable compilation-phase output will be written.

PFCC (carriage control)
PFLL (line limit)
PFLP (lines per page)
PFLL (page limit)

See subsection (a), above, for details concerning these parameters.

Q1=filename ('Q1' file)

Specifies the name of the 'preliminary' code file passed from the SETL compiler's parse phase to its optimisation or code generation phase. See remarks concerning this parameter in subsection (b), above.

Q2=filename ('Q2' file)

Specifies the name of the 'interpretable' code file passed from the SETL compiler's code generation phase to the run-time support phase when the SETL system is being run interpretively. See Section XXX for an explanation of the role played by this file.

(d) Run-time support library options

ASSERT=1/2 (assertion switch)

This parameter can have 0, 1, or 2 as its value. These values have the following significance:

ASSERT=0 Evaluates all Boolean conditions occuring in ASSERT statements, but does not test their values. (Note that evaluation of these conditions may trigger side-effects essential to the proper functioning of the program being run.)

ASSERT=1 Evaluates and tests all assertions. Assertions which fail yield a run-time error.

ASSERT=2 Evaluates and tests all assertions. A message is printed for each assertion which evaluates to TRUE. Assertions which fail yield an error.

H=0/1 (Heap size)

This command parameter, which specifies the initial (virtual) memory length used during a SETL run, is significant to both the code generation phase of the compiler and to the run-time support library. See the account of this parameter in the preceding subsection for additional details.

LCP=0/1 (list execution time parameters)

The option LCP=1 causes the values chosen for standard control card options to be listed on the output file at the start of SETL program execution.

LCS=1/0 (list execution statistics)

The option LCS=1 causes various standard statistics collected during execution of your program to be printed at the end of SETL program execution.

Q2=filename ('Q2' file)

Specifies the name of the 'interpretable' code file passed from the SETL compiler's code generation phase to the run-time support phase when the SETL system is being run interpretively. See the remarks concerning this parameter in subsection (c) above.

REL=0/0 (run-time error limit)

The value of REL specifies the run-time error limit. If more than the specified number of errors are detected during SETL execution, then execution terminates.

SB={ }/<<>> (set brackets)

The value of SB specifies the characters to be used for printing set brackets.

SNAP=0/1 (snap dump switch)

The SNAP=1 option causes an abbreviated dump of recent variable values to be produced when a run-time error is detected. Specify SNAP=0 to suppress this dump.

STRACE=0/1 (Statement trace)

Selecting the option STRACE=1 causes production of a dynamic trace giving the statement number of each statement executed. This option should be used cautiously, since it tends to produce very voluminous output. The statement numbers used are those which appear on the parse phase output listing.

TB=[ ]/( ) (tuple brackets)

The value of TB specifies the characters used for printing tuple brackets.

(e) Other command parameters used for system checkout and maintainance.

In addition to the command parameters listed above, the SETL compiler recognises various other parameters, which are provided for purposes of system checkout and maintainance and are not needed in normal use. Note, however, that you must avoid using the names of these parameters to designate other quantities which your SETL program will read from the control card using GETIPP and GETSPP.

We list these special parameters with brief indications of their function, but give no details concerning them. For more information about these parameters, consult the SETL Maintainance Manual. The maintainance facilities listed above are activated by the SETL command parameters listed. A second family of maintainance facilities are activated by inserting special statements of the form

DEBUG dopt1,...,doptk;

into the text of a SETL program being compiled. Here, doptl,..,doptk should be a list of keywords designating debug options. Some of these debug options refer to the parse phase of the SETL system, others to the semantic analysis, code generation, or execution phases. Since the ordinary user will have little reason to concern himself with these options, we list them here in abbreviated fashion only; see the SETL Maintainance Manual for more informtion.

(i) Parse phase debug options:

PTRM0	disable macro-processor trace
PTRM1	enable macro processor trace
PTRP0	disable parse trace
PTRP1	enable parse trace
PTRT0	disable token trace
PTRT1	enable token trace
PRSOD	list tokens corresponding to loops and IFs still pending
PRSPD	list polish and xpolish tables
PRSSD	list symbol table

(ii) Semantic analysis phase options:

STRE0	disable entry trace
STRE1	enable entry trace
STRS0	disable trace of operator argument stack
STRS1	enable trace of operator argument stack
SQ1CD	list Q1 code
SQ1SD	list semantic analysis phase symbol table
SCSTD	list stack used for processing control structures and other nested constructs

(iii) Code generation phase options:

CQ1CD	list Q1 code
CQ1SD	list code generation phase symbol table
CQ2SD	list generated Q2 code

(iv) Execution phase options:

RTRE0	disable trace of entry to run-time library procedures
RERE1	enable trace of entry to run-time library procedures
RTRS0	disable statement number trace
RTRS1	enable statement number trace
RTRC0	disable code trace
RTRC1	enable code trace (The code trace prints each internal 'Q2' instruction as it is interpreted.)

RTRG0	disable garbage collector trace
RTRG1	enable garbage collector trace
RGCD0	disable dynamic storage dumps during garbage collection
RGCD1	enable dynamic storage dumps during garbage collection
RDUMP	dump dynamic storage to file (The file to which an image of dynamic storage is written is specified by a control card parameter: DUMP=filename. The auxiliary maintainance program DMP reformats this file in a readable form.)

8.6 Exercises

Ex. 1 Write a program which will read a sequence of lines constituting an English language text, and print it out after eliminating all multiple blanks and assuring that every punctuation mark (other than hyphen) is followed by exactly one blank.

Ex. 2 The position on a chessboard is defined by a mapping f which sends every square [i,j] occupied by a piece into the name of the piece occupying it. Pieces are designated by their names, e.g. 'pawn', 'king', 'queen', etc. White pieces are designated by lower-case names, e.g. 'pawn'. Black pieces are designated by upper-case names, e.g. 'PAWN'. Write a procedure which prints an attractive visual display of the board position.

Ex. 3 Given the representation of chessboard position described in Exercise 2, write procedures which will

(a) return the set of all moves possible for white or black;
(b) return the set of all white or black pieces threatened with capture;
(c) return the set of all squares attacked by white or black pieces.

Ex. 4 When crucial items of information like invoice or customer numbers need to be keyed into a computer system, the possibility of keypunch error is often quite alarming. To prevent such errors, one often adds 'check characters' to the item being keyed in. Such check characters allow miskeyed items to be detected in most cases. If any alphanumeric check character from 0..Z can be used, the following is a convenient way of assigning check characters:

(a) Number all alphanumeric characters, assigning them values lying in the range 0..35.

(b) Go through the characters of the item to be keyed in, from right to left. Multiply the number associated with the j-th item by j, and sum all the resulting integers. Reduce the sum modulo 37, to obtain an integer n.

(c) The check character is the character corresponding to n, or is Z if n is 36.

Ex. 5 Write a key-entry verification program. This program should begin by reading a file F of lines that are to be verified. Those same lines should then be re-entered at the terminal. If a line L re-entered has exactly the same form as the corresponding line in the file F, then L should simply be displayed. If not, then the terminal should (if possible) emit a warning audio signal, and the line L0 present in the original file F should be displayed along with the L1 just entered. Characters which need to be replaced in L0 to make L match L1 should be marked by displaying appropriate replacement characters under L1; characters which need to be deleted should be indicated by displaying a double quote character under L1. If one or more characters need to be inserted, they should be displayed in a vertical column under the character of L1 after which they need to be inserted. In the event of a difference, the user ought to have the following three options:

0: accept L0 as correct
1: accept L0 as correct
2: re-enter line, and repeat the check.

Ex. 6 Write a procedure which can be used to display 'menus' on the screen of an interactive terminal. The parameter passed to this procedure should be a tuple [s1,s2,...] of strings. As many of these strings as will fit on the screen should be displayed, each accompanied by a number. The user should then type one of these numbers to select the desired item, and the procedure should return the number of the item selected. If something illegal (e.g an out-of-range integer) is typed, the procedure should return OM. The display you use should be neatly formatted, in multiple columns if possible, to display as many items as possible on the screen without giving the screen a cluttered appearance. If not all items will fit on the screen, the message

PRESS RETURN KEY TO SEE ADDITIONAL CHOICES

should be displayed at the bottom of the screen. (Of course, the feature described by this message should be implemented in a fool-proof manner.)

Ex. 7 Write a program that will read SETL source text and count the number of comments in it. A record should be kept of the number of 'short' comments (which occupy just one line and are followed by a line containing code text), and the number of 'long' comments (which occupy several successive lines on which only comments appear.) Two counts of long comments should be kept, namely the number of long comments 2-4 lines in length, and the number of long comments five or more lines in length. You should also count the total number of lines in the program. Note that every comment starts with a dollar sign ($) character, but that such a character only starts a comment if it is not part of a quoted character string. For example, the first character of the first comment in

```
x:='I often think of $''s,$''s,$''s'; $ Not really!
```

is 'N'. Be sure to handle this rule properly.

Ex. 8 When punched cards are used to transmit information to a computer system, it is sometimes convenient to pack information densely onto them, without blank spaces between successive information fields. In this case, the size of each information field in a line of characters must be known in advance. Write a procedure P whose two inputs are a string s of exactly 80 characters representing a punched card being read, and a tuple t representing the 'format' of this string, i.e. the size and nature of the successive information fields in it. Each component of t should have the form [n,k], where n is a positive integer designating the number of characters in a particular subfield of s, and k is one of 'I' (integer), 'S' (string). The procedure P should return a tuple of converted values, with each value of improper type represented by OM.

Ex. 9 To develop a KWIC (or 'key word in context') index for a body T of text, one proceeds as follows:

(a) A collection of keywords is given.

(b) The text T consists of a collection of paragraphs, each headed by a 'paragraph designator' at most one-third of a line long.

(c) The paragraphs constituting the text T are scanned for occurences of any of one the keywords. Whenever a keyword is found, a portion L of the line in which it occurs, two thirds of a line in total length, is kept, with the keyword as close to the middle of this line section as possible. (Words from preceding or following lines are included if necessary.)

(d) A designator of the paragraph containing the line is concatenated to L, and the resulting string is added to a collection s of strings.

(e) When the whole of T has been scanned, the set s of all lines collected is alphabetised according to the keyword each L contains, and is printed in alphabetical order, with keyword capitalised.

Write a program which generates KWIC indices of this kind.

Ex. 10 Write a program P which can be used to scan a mass of English-language text T, counting the frequency of all letter pairs encountered. Use P to scan a few paragraphs of text. Count the total number of pairs encountered and the total number of different pairs. Draw a graph relating number of different pairs encountered to the total number of characters scanned, and use this to estimate the number of different character pairs that you would encounter if you scanned the whole Encyclopedia Brittanica.

Ex. 11 A telegram is transmitted as a single string of characters, with words separated by blanks but the end of each line marked by a dummy word 'ZZZZ'. Write a program which will count the number of words in the actual telegram. Words with more than eight letters in them are to count as two words.

Ex. 12 Write a program that will read three strings s1,s2,s3 and then determine whether s2 occurs as a substring of s1 after all characters belonging to s3 have been eliminated from s1.

Ex. 13 A spelling error program is one which reads an input text T and produces a list of all the words in T which appear to be misspelled. One way of making this check is to test each of the words in T to see if it belongs to a standard dictionary D of properly spelled words. Write such a spelling error program. Your program should read two files: one the file T to be checked, which is given as a sequence of text lines; the other a dictionary D, also assumed to be a sequence of lines, each line containing several dictionary words, separated by blanks. The file T can contain capitalised words. The output produced should be a formatted display of all presumably misspelled words.

Ex. 14 Assuming that the spelling error program described in Exercise 13 is to be run interactively from a terminal, improve it by adding the following features. The program should begin by querying the user for the names of the files T and D. Then it should read and analyse these files as in Exercise 13. The misspelled words in T must then be numbered and displayed on a terminal. After this the program should accept a sequence of commands of the form

```
n1correctspelling1/n2correctspelling2/
```

where each nj is the number of a misspelled word and correctspellingj is its correct spelling. This sequence of commands is terminated by a command of the form

```
STOP,
```

after which the program should query the user for the name of an output file F into which a corrected version of the input T is to be written. All occurences of misspelled words in T for which correct spellings have been supplied should be corrected, and the corrected text which results written out to F.

Ex. 15 'Piglatin' transposes words by moving all initial strings of consonants to the end of the word and adding 'ay'. If a word begins with a vowel, one simply adds 'ay' to it. The word 'a' is changed to 'an'. For example, the Piglatin translation of 'John bought a car from Irene' is 'Onjay oughtbay an arcay omfray Ireneay.' Write a program which will read an input text and translate it into Piglatin. Your program should handle capitalisation and punctuation correctly.

Ex. 16 Write an interactive program which could be used by a teacher to maintain class grade records. The specifications for this program are as follows. It maintains a list of all the students enrolled in a class. Each student name is mapped into a directory record giving the

student's address, telephone number, and any desired additional textual information concerning the student. Each student name is also mapped into a tuple giving the student's grade on a sequence of homework exercises and examinations. Finally, each assignment or examination number is mapped onto a line of text describing the assignment or examination.

The system should accept at least the following commands:

E/student-name	enroll student with given name
D	display numbered table of all students
IE	set up for entering new textual information concerning students
IA	set up for entering textual information concerning assignments
G/n	set up for entering new grade information for assignment n
D/n	display all information concerning student n
n/line-of-text	enter line of text to information record of student n, or enter comment about assignment n, or enter grade for student n
DA	display information concerning assignments
DA/n	display information concerning n-th assignment, including average grade, highest and lowest grade, number of students in each grade quintile, and names of students who have not yet completed assignment.

What other commands would be useful? Design at least three such commmands, docment them, and include them in your implementation.

Ex. 17 The 'game of life', invented by James H. Conway, models certain elementary biological phenomena. The simulation it embodies takes place on an n by m board. For definiteness, we will suppose that n=m=20. At every step of the simulation, every square on the board is either occupied (by an 'organism'), or empty. Given such a configuration, the next 'generation' of organisms (i.e. the board configuration at te next step) is determined by the following rules:

(a) If a square is empty but has three or more full neighbors, it will become full (since an organism will be 'born' in it.)

(b) If a full square has four or more full neighbors, it will become empty (since the organism in it will 'smother'.)

(c) If a full square has no neighbor or just one neighbor, it will become empty (since the organism in it will 'starve'.)

Write a interactive program which reads an initial board configuration, and then simulates its evolution for a given number of steps.

Ex. 18 Write an interactive program for use by bank tellers. This program is to maintan a map which sends each client of the bank into his current balance, and another similar map which sends each client into his name, address, and phone number. (Clients are identified by unique 'account numbers' issued by the bank). Finally, each client is mapped into a maximum allowed 'line of credit' and to the sum currently drawn

against this line of credit.

TO BE CONTINUED

Ex. 19 Write an interactive program that plays the game HANGMAN. Your program should read the date and time, and use this to select a word at random from an internally stored collection of 100 words. The player should then be asked to guess the word, one letter at a time. Each guessed letter present in the word should elicit a display showing all letters guessed so far. If the number of incorrect guesses rises to half the number of letters in the word being guessed, the player loses, and the word should be revealed. Try to write an entertaining program. Your program should keep score of the number of games won and lost.

Ex. 20(a) Many large software systems include interactive 'HELP' subsystems which, when entered, allow a tree of helpful information to be traversed. The aim is to make it easier for the system user to locate information which he may require in order to use a system successfully. Use an appropriate variant of the 'menu' procedure in Exercise 5 to implement such a HELP system. When invoked, the HELP system should begin by reading a file describing part of a graph of nodes, each node representing a state which the system user can reach during his browsing. In each state, the system should display a short paragraph of helpful information and a menu of available subitems. The information which the HELP system needs should be divided into a set of files, each few hundred lines in length, which can be read seperately as requested by the user. Each such file will contain two maps, which we shall call nodes_map and files_map. Nodes_map has the following format:

```
{[help_graph_node,[display_paragraph,subnode_menu]}
```

The display_paragraph is a tuple consisting of a sequence of lines (strings) to be displayed when help_graph_node is reached. Subnode_menu is a tuple of help_graph nodes, to be displayed as a menu. By selecting an appropriate item of this menu, the user chooses the HELP system node to which he wishes to advance next. By typing '-', the user retreats to the last HELP system mode previously examined.

The other map, files_map, which is available in a pre-established file which can be read by the HELP system, simply maps each subnode x referenced by a given file to the name of file which contains the display_paragraph and subnode_menu information for x.

(b) In order to use the HELP program we have just described, you will find it convenient to design and implement a 'HELP setup' program which can read a file of text containing all the paragraphs and defining all the menus which will appear in the data structures described in (a) above. This file should also define the manner in which all this information is to be divided into the smaller files which the HELP system will use. Design and implement this 'HELP setup' package. Your 'HELP setup' program should verify that the information passed to it is internally consistent.

Ex. 21 Write a record-keeping system suitable for daily use in a library. The system should read files of instructions, and generate

various outputs. Each transaction handled by the system starts with a command line whose first two characters are '**'. The transactions handled are as follows:

```
**E card_number
name
address
telephone_number
```

This transaction enrolls a new subscriber, assigning him the indicated library card number. The card number must be unique, or the transaction will be rejected. The subscriber's date of enrollment is internal.

```
**C card_number address
new address
```

This transaction changes the address recorded for a given subscriber. Similar transactions which change the name and telephone number provided for a given subscriber should also be provided.

```
**L card_number
```

This transaction lists, in appropriately sorted order, the information available for a given subscriber, including all books currently charged to the subscriber, with dates of withdrawal, and number of books borrowed in current calendar year.

```
**A
```

This transaction produces an alphabetised list of all subscribers, with addresses and telephone numbers.

```
**B book_number card_number
```

This transaction charges a book to a customer.

```
**R book_number
```

This transaction notes that a book has been returned.

```
**Q book_number
Title, Author
Publisher, Publication date
```

This transaction notes the aquisition of a new book, and assigns it a book number. The book number must be unique or the transaction will be rejected. The date of acquisition is noted, internally.

Books can be borrowed for two weeks. Books not returned within a two week limit are considered to be overdue. When run, the library record system should produce a warning letter to all subscribers holding overdue books. However, no subscriber should get such a letter more often then once a week. In this dunning letter, books should be listed by title and author. Books for which a previous notice has been sent should carry the additional legend 'SECOND NOTICE', 'THIRD NOTICE', or

'GROSSLY OVERDUE'.

The transaction triggered by
**D

should produce an alphabetised list of all subscribers holding unreturned books for which more than two notices have been sent, with an indication of the number of 'THIRD NOTICE' and 'GROSSLY OVERDUE' books they are holding.

Can you design, and implement, any other useful feature for such a system?

Ex. 22 Write an interactive 'daily reminders' program. This program reads a file of one-line messages, each tagged with a given date, and displays them. Messages displayed are also numbered. The system is to handle the following commands.

-	(display all reminders remaining from past days)
+n	(display all messages relating to n days from today)
+	(display all future reminders)
?n	(delete reminder n).

Define and implement commands for dealing with the situation in which unmodified execution of a command would display too many messages to fit all at once on your terminal.

Ex. 23 Write an election forecasting program. The base data for the program should be a map sending each voting precinct into its total of Democratic and Republican votes in the last election, into its state, and into its type: urban, inner city, suburban, and rural.

The program will be run every thirty minutes on election night. As returns come in from various precincts, these will be compared with the returns from same precincts in the last comparable election. If the Democratic and Republican percentages reported for a given type of precinct in a given state are D and R, while the prior Democratic and Republican percentages for the same precinct were d and r respectively, then the Democratic (resp. Republican) gain can be estimated as the quotient D/d (resp. R/r.) Use these gain gactors to extrapolate the vote for all precincts of the same general character that have not yet reported.

Ex. 24 It is sometimes hard to use an operating system's interactive facilities without a manual at your elbow, for two reasons: (a) the system provides many facilities and it is hard to remember them all (b) most operating system commands have numerous parameters and options, whose names and effects are hard to remember.

Write an operating system command assistance program which will make it easier to compose operating system commands. When invoked, this program should display a numbered menu of all available commands, with one-line comments concerning the purpose of each. When one of these commands is selected (by number) a numbered menu of obligatory and optional command parameters and options should be displayed, with a set of one-line comments on the form and effect of each parameter. The user should then be able to enter parameter values and select options, by number, either on a single line (separated by blanks), or on several successive lines. When he is finished, the command line that he has composd should be displayed. Use an appropriate variant of the 'menu' procedure described in Exercise 5 to build up this program.

Ex. 25 Extend the program described in Ex. 24 so that its user can actually issue the command that he has composed. (Use the HOST feature described in Appendix XXX for this purpose.) The user should also be able to re-start entry of parameters and options so that he can modify any parameters and delete any options with which he is not satisfied.

Ex. 26 Extend the program described in Exercises 24 and 25 so that it allows command language programs to be composed and saved for subsequent use.

Ex. 27 Write an interactive 'perpetual calendar' program. This should handle the following commands:

month/year (displays calendar for the requested month)

day/month/year (displays calendar for the week containing the specified day.)

Ex. 28 If person A makes a taxable payment to person B, he informs the Internal Revenue Service of this fact, giving the amount of the payment and the social security number of person B. Person B is then expected to file a report stating his total income. Write a program which will read a file of lines, each having either the form

PAYMENT (social security number of recipient) (amount)

or

INCOMEREPORT (social security number of person reporting) (amount)

and will then detect all persons who seem to be concealing more than $200 in income. A list of persons, with the persons concealing the largest amounts of income coming first should be printed.

Ex. 29 A company bills its customers on the fifteenth of each month. Bills fully paid within 14 calendar days of their receipt are granted a 1% discount, bills fully paid within 30 days of their receipt are charged their face amount. Other bills pay a 2% per month interest charge. Write a program which will read two files of records, the first having the form

BILL bill_number customer_number amount date

PAYMENT bill_number customer_number amount date

The program should print a list of all bills for which full payment has not been received, with a statement of the amount still owing on the bill. The 'date' entry on each line of the file will be a string: for example, Jan. 9, 1980 would be represented as 1/9/80.

Ex. 30 After studying the 'eight queens' program presented in Section 8.2.1 write a modified, more efficient backtrack program for solving the eight queens problem which places queens one after another in appropriate rows of successive and exploits the fact that at most one queen can be placed in each row. Modify this program further so that it produces all possible solutions of the eight queens poblem, but suppresses configurations that can be obtained from a known solution by reflecting the chessboard through one of its axes of symmetry.

Ex. 31 Write an 'n-bishops' program which will place as many bishops as possible on an 8 x 8 chessboard in such a way that no two bishops attack each other. (Hint: For a somewhat more efficient program than would otherwise result, work thru the diagonals of the board in succession, exploiting the fact that no two bishops can be placed on the same diagonal.)

Ex. 32 Modify the tiling program given in Section XXX so that it works with rectangular rather than square tiles, where each rectangular tile can be placed either horizontally or vertically, i.e. can be placed in one of two orientations differing from one another by 90 degrees.

Ex. 33 Write a procedure which uses backtracking to calculate and return the power set POW(s) of a given set s.

Ex. 34 Let G be a graph, given as a set of ordered pairs, each representing an edge of the graph. A topologically sorted order for G (c.f. Exercise XXX) is an ordering of its nodes such that each edge of G goes from a lower-numbered to a higher-numbered node. Write a program that reads in a graph G and then uses backtracking to generate all topologically sorted orders for it.

Ex. 35 It is often straightforward to eliminate backtracking from simple backtrack programs by using recursion instead of backtracking. When this is done the information required for backtracking is saved in successive 'incarnations' of a recursive procedure, -OK- is replaced by a recursive call which creates a new 'incarnation' rather than a new backtrack 'environment' (cf. Section XXX), and -FAIL- is replaced by a recursive return.

Apply this idea to develop recursive routines which solve the eight-queens and tiling problems described in Section XXX.

Ex. 36 Build up an inclusion library containing the following procedures: mergesort (Section XXX), polynomial package (Section YYY). Using this inclusion library, write a program which reads in a collection of pairs of polynomials represented as vectors, multiplies them, sorts the resulting product polynomials P into decreasing order of the values P(1), and then prints them out. Polynomials should be

printed in something like their standard representation, e.g. the polynomial read in as [1,2,0,3] should be printed as 3**x**3 + 2*x+1.

Ex. 37 A machine tool company manufactures various kinds of tools, each of which consists of several kinds of parts manufactured by various of its departments. Information concerning parts requirements is stored as a map

```
{[tool_name,parts_map],...}
```

where each tool_name is the name of a particular tool that the company manufactures, and each parts_map is a mapping from the name of each part used in the manufacture of the tool to the quantity of this part required and the name of the department responsible for manufacturing the part. (Thus parts_map has the form

```
{[part_name,[number,department]],..})
```

Write a program which will read a list of orders, each having the form

```
order_name,tool_name,quantity_ordered
```

and make up a list of parts orders arranged by department. Each parts order generated should be headed by the current date and a department name, and should then consist of succesive groups of lines, arranged in alphabetical order by part name. Each such group should start with a line having the form

```
part_name,total quantity needed
```

and continue with a sequence of lines having the form

```
order_number, quantity in order
```

These latter lines should be arranged by order_number.

The parts order to be sent to a given department can extend over many pages. Every page of this order must be headed by the current date and the appropriate department name, and also by an appropriately positioned caption reading 'Page j of n', where n is the total number of pages going to a given department and j runs from 1 to n. The parts order to be sent to a given department should always start at the top of a new page.

Ex. 38 'Encoded arithmetic' puzzles are a common form of mathematical recreation. In puzzles of this kind, digits are represented by letters of the alphabet, and then an arithmetic relationship is written: for example SEND + MORE = MONEY. To solve the puzzle, one must determine the digit value of each character. Such problems can of course be solved by a backtracking search through all possible assignments of digits to letters, but the following remarks suggest a more efficient approach:

(a) In each digit position, a carry is either present or absent. Depending on the assumptions which we make about carries, each digit position in an enciphered sum leads to one of several equations. E.g., if in the example SEND + MORE = MONEY we assume that carries are present

in the second and third digit positions from the right, then we must have N + R + 1 = E + 10 i.e. N+R=E+9.

(b) These equations can be used to eliminate as many variables as possible. For example, since the preceding example involves 8 letters and generates 5 equations, we can solve for the digit values of all 5 letters in terms of only three of them.

(c) A solution can then be obtained by backtracking through all possible values for the uneliminated letters, and all possible carry patterns. (In the example considered, this will mean that 32,000 possibilities are examined.)

Write a backtracking program along these lines. Your program should be able to solve any encoded addition problem. It should generate all possible solutions. Use your program to solve SEND + MORE = MONEY, and DONALD + GERALD = ROBERT. What modifications to your program are necessary if it is to solve encoded arithmetic puzzles for addition modulo 8?

Ex. 39 This exercise will describe a relatively elaborate page-oriented output facility, which you are asked to program. Your program should be written as a single MODULE.

The output facility to be programmed will allow a page, which is to be filled with elegantly formatted string text, to be divided into nonoverlapping, named areas, which can then be written separately. To define such a page layout, a multi-parameter procedure LAYOUT, with parameters like those shown in

```
LAYOUT(field_name_string,field_descriptor_1..,field_descriptor_n);
```

is used. The field_name_string parameter is a string, consisting of blank-separated names, each of which names one of the fields whose position and size is defined by a subsequent field_descriptor. The j-th name and the j-th field descriptor correspond to each other.) The nature of any field can be further qualified by appending a qualifier to its name. Attaching a qualifier .R to a field name specifies that incomplete lines written to this field (see below) are to be right-justified; similarly, the qualifier

.L specifies left-justification, and the qualifier .C specifies centering.

Each field_descriptor has the form

```
[starting_line, starting_position, width, height]
```

Here, starting_line indicates the line number at which a given field is to start (lines are numbered sequentially down the page, beginning at line 1), and starting_position indicates the horizontal position (numbered from position 1 at the extreme left) at which the field is to start. The two final quantities -width- and -height- define the horizontal and vertical dimensions of the field.

The LAYOUT procedure returns TRUE if it detects no inconsistency (e.g. overlapping fields) in the requested layout; but FALSE otherwise.

After defining the layout of fields on a page usng the LAYOUT procedure, one can write to any or all of these fields, using a call

```
WRITE(field_name_string, s1, s2, .., sk);
```

Here, field_name_string consists of a blank-separated sequence of field names, to which the remaining strings s1, s2,..sk will be written in sequence. Any field_name in this field_name_string can be qualified by appending one or more characters '*' to it; a single asterisk terminates the current line of the field (moving down one line in this field) and additional asterisks skip one line each.

The quantity of information already written to a given field, or to the whole page, can be sensed by invoking the function

```
AMOUNT(fieldname,s).
```

Here the parameter s indicates what is wanted, specifically s='LINES' calls for the number of the last-written line of the indicated field, s='CHAR' calls for the number of characters already written to this line,, and s='DESCRIPT' retrieves the descriptor of the field. The simplified invocation

```
AMOUNT( )
```

returns the number of the last line written to any field.

Finally, the call

```
OUTPUT( )
```

prints the page that prior calls to LAYOUT and WRITE have built. Moreover, it is legal to invoke LAYOUT several times before OUTPUT is called. This allows material to be written to a single page using several successive layouts.

As you program this package of procedures, you will become aware of various incompletenesses in the preceding specificatons. Resolve all these ambiguous points in tasteful ways, and then document your decisions carefully, so as to create a detailed user's manual for the 'page layout facility' that you will create.

Ex. 40 Design, and implement, various useful extensions to the page layout facility described in Exercise 39. For example, you may want to allow area names to be qualified with .J in a LAYOUT call, thereby indicating that material written to an area is to be printed in right-and-left justified form.

Ex. 41 Use the page layout facility described in Exercises 39 and 40 to print out the title of a book, and to print the first page of chapter one of the book, wth appropriate chapter headings, and with the body of the first page in a two-column format. This first page should include

at least one imbedded table.

$

CHAPTER 9

PROGRAMS, MODULES, LIBRARIES, AND DIRECTORIES

Chapter IX. Programs, Modules, Libraries, and Directories: Extended Structuring Constructs for Large SETL Programs. Remarks on the SETL run-time system.

We noted in Sections 7.1 and 7.9 that for clarity and to avoid error it is important to divide any program consisting of more than a few dozen lines into logically independent paragraphs, each of which performs a well defined function in a manner free of close involvement with the details of other code paragraphs. We also noted that the procedure, function, CASE, and refinement constructs of SETL are the main tools which it makes available to aid this kind of 'paragraphing'. Together these tools serve as reasonably adequate extended structuring constructs, which make it easy to divide a long program into parts called modules and libraries. If such a division is made, two supporting code sections, one called the (main) program of the overall text being constructed, the other constituting an overall directory of the text are also required. In normal usage, each module and library will consist of several dozen procedures and functions, and will contain declarations of all ('module-global') variables directly accessible to more than one procedure of the module or library; the directory, which consists of declarations only, will indicate which of the procedures in each module are available for use in other modules, and will declare a set of 'program-global' variables available to all procedures in all modules.

In this section, we will describe SETL's extended structuring concepts systematically, and will illustrate their use.

Chapter Table Of Contents

9.1 Textual Structure of Complex Programs

A program text can either be a simple program like those described in the preceding chapters of this book, or can be a complex program. A simple program consists of an optional sequence of declarations, a main program part, and a collection of procedures and functions: the role which all of these structures play has already been described in previous chapters. (See especially Chapter IV.) A complex program, which has a richer structure, consists of the following items in sequence:

(1) a single directory, followed by

(2) a single program unit, followed by

(3) a collection of one or more modules and libraries.

We begin our detailed account of this family of constructs by describing the structure and purpose of module and library units. Each module consists of the following items in sequence.

(i) A header line.

(ii) Optionally, a collection of one or more library items.

(iii) Optionally, an access specification. If present, this will describe the relationship of the module M to the other modules present in the same complex program.

(iv) Optionally, a sequence of declarations. If present, these will define variables and constants globally accessible to all the procedures in the module M, will call for certain initializations, and will sepcify the manner in which particular variables are to be represented.

(v) A sequence of one or more procedures (and functions).

(vi) A trailer line, which closes the module.

The following example shows all these features except (iii):

```
MODULE logic_analyzer - syntactic_decomposition; $ header line
LIBRARIES lexical_analysis, error_reports;       $ library item
LIBRARIES error_tracing, error_reporting;        $ additional
                                                     library item
VAR                                                $ declaration of
   Formula_grammar, Expression_grammar;            $ 'module-global'
                                                     $ variables
   Analysis_stack;

VAR                                          $ additional declaration
   Parse_status;                           $ of 'module_global' variable

CONST                                        $ constant declaration
   Expr = 1, Term = 2, Factor = 3;
```

```
INIT                                          $ initialization declaration
   Analysis_stack := [], Parse_status=2*3;

REPR                                          $ representation declaration
   Formula_grammar, Expression_grammar: $ (see Chapter 10 for an
      SMAP(INTEGER) TUPLE(INTEGER);          $ explanation of
   Analysis_stack: TUPLE(INTEGER);          $ representation declarations)
END REPR;

PROC parser(x);                               $ first procedure of module
      ... $ body of procedure
END PROC parser;

PROC special_actions(y),                     $ second procedure of module
      ... $ body of procedure
END PROC special_actions;

...                                           $ additional procedures of
...                                           $ module would follow here
END MODULE logic_analyzer - syntactic decomposition; $ trailer line
```

This example illustrates the following general rules:

(i) The header line of a module consists of the keyword MODULE, followed by a pair of identifiers separated by the sequence space-dash-space. The first of these identifiers is a directory-name; it names the directory which comes first in the (complex) program containing the modules and must be the same for all modules in a program. For example, the module shown above would have to follow a directory whose header line was

```
DIRECTORY logic_analyzer;
```

and the other modules in this same (complex) program would have header-lines like

```
MODULE logic_analyzer - propositional_calculus;
   ...
MODULE logic_analyzer - predicate_calculus;
   ...
```

etc.

(ii) Each member of the optional sequence of library items which then follow in the module M consists of the keyword LIBRARIES, followed by a comma-separated sequence of library identifiers, each of which names one of the libraries in the complex program (see below) which the module M item needs to use.

(iii) The (optional) access specification which can then follow is described later in this section. See 'Directories', below.

(iv) The optional declarations which follow after this have the same structure as the global declarations included in a simple program. VAR, constant, INIT, and REPR declarations are all allowed, and can be given in

any order. VAR declarations appearing in this position within the module M specify variables having module-global namescope, i.e. variables accessible to all the procedures in M (but to no other procedures).

Libraries have essentially the same structure as modules, except that the header line of a library module begins with the keyword LIBRARY, which is followed by a simple identifier (the library name) rather than a hyphen-separated pair of identifiers, as a standard MODULE would be. Moreover, none of the procedures in a library can either access variables or invoke procedures declared outside the library. We can therefore say that, whereas modules constitute the chapters of a complex program outside of which they are not likely to be used, libraries contain self-stand ing collections of utility routines and are likely to be used in many different programs.

The single program unit allowed in a complex program has much the same structure as a module, except that before the collection of routines which it contains there must occur one or more statements constituting its main program. Execution will then begin with the first statement of this main program.

More specifically, a program unit consists of

(i) A header line, consisting of the keyword PROGRAM, which must be followed by the appropriate directory name (see (i) above) and then by the name of the program unit itself, these two items being separated by the sequence blank-hyphen-blank, as in

```
PROGRAM logic_analyzer - main;
...
```

(ii) Optionally, a collection of one more library items.

(iii) Optionally, an access specification (just as in a module).

(iv) Optionally, a sequence of declarations (VAR, CONST, INIT, and REPR declarations, as in a module or a simple program).

(v) The 'main program', i.e. a sequence of one or more executable statements.

(vi) An (optional) collection of one or more procedures.

(vii) A trailer line, terminating the program unit.

Note again that if optional items (ii) and (iii) are omitted, we will have exactly a simple program of the stand-alone sort that could be used without a directory.

Next we describe the structure of a directory; this will also explain the structure and purpose of the directory item (cf. (iii) above) that can be included in any module, library, or program unit. A directory consists of

(i) A header line;

(ii) Optionally, a set of declarations (VAR, CONST, INIT, and REPR declarations, exactly as in a module or library);

(iii) A single PROGRAM descriptor;

(iv) A sequence of MODULE descriptors, one for each module which follows the directory;

(v) A trailer line, which terminates the directory.

These objects are subject to the following general rules:

(i) The header line of a directory consists of the keyword DIRECTORY, followed by an identifier which names the directory, as in

```
DIRECTORY logic_analyzer;
```

As already stated, this identifier must be repeated in in all the modules, and also in the main PROGRAM unit which follows the directory

(ii) The optional VAR and CONST declarations occurring in the directory define the names of program-global variables and constants accessible to (the main program and) all procedures (other than library procedures) in the complex program in which the directory appears. The optional INIT declarations appearing in the directory serve to initialize these program-global variables, and any REPR declarations appearing in the directory serve to defie representations for these variables.

(iii,iv) The program descriptor and module descriptors which come next serve to define the manner in which the program unit and modules which follow the directory are allowed to access the global variables declared in the directory, and also determine which procedures in which modules can be invoked by procedures in other modules. The syntactic form of these descriptors is

(for a program descriptor)
PROGRAM directory name - program name: access specification;

(for a module descriptor)
MODULE directory name> - program name>: access specification;

That is, the first part of each such program or module descriptor is identical with the header line of the program or module it describes; but this first part must then be followed by an access specification. Such an access specification has the following components:

(a) an (optional) item of each of four possible types:
READS, WRITES, IMPORTS, and EXPORTS items

(b) an (optional) REPR declaration.

A READS (resp. WRITES), item consists of the keyword READS (resp. WRITES), followed either by a list of names of program-global variables and constant s, or by the keyword ALL. This is shown in

```
READS ALL;
WRITES Phase, Subphase;
```

These items serve to define the program-global variables and constants which are read (resp. written) by one or more of the procedures in some MODULE (or in the main program of a PROGRAM unit) of a complex program.

An IMPORTS item lists and describes all the procedures defined elsewhere which are used within a unit. It consists of the keyword IMPORTS, followed by a sequence of procedure descriptors, each of which is identical to the header line of the procedure being described (but omitting the keyword PROC or PROCEDURE). Procedure parameters which are read-only, write-only, or read-write must be declared in the procedure descriptor of an IMPORTS statement in precisely the same way as they are declared in the header line of the corresponding procedure, i.e. must use RD, WR, or RW, precisely where these occur In the header line. This is shown in the following examples:

```
DIRECTORY logic_analyzer;
...

MODULE logic_analyzer - propositional_calculus;

    IMPORTS b1(x),                          $ a one-parameter function
            b2(RW x, RD y)
            b3(x, y, z(*));                  $ procedure with variable number
    ...                                     $ of parameters

MODULE logic_analyzer - predicate_calculus;

    IMPORTS a1(x,y,RW z),
            a2;                             $ parameterless procedure
...
END DIRECTORY;                              $ directory trailer line

MODULE logic_analyzer - propositional calculus;
    ...

PROC a1(x,y,RW z);                          $ note correspodence with
...                                         $ preceding declaration
END PROC a1;

PROC a2;                                    $ parameterless procedure
...
END PROC a2;

...
END MODULE logic_analzer - propositional calculus;   $ trailer line

MODULE logic_analyzer - predicate_calculus;
...
```

```
PROC   b1(x);                          $ a one-parameter function
...
END PROC b1;

PROC b2(RW x, RD y);                   $ note correspondence
...                                    $ with preceding declaration
END PROC b2;

PROC b3(x,y,z*);                       $ procedure with variable number
...                                    $ of parameters
END PROC b3;

...
END MODULE logic_analyzer - predicate_calculus;
```

An EXPORTS item lists and describes all the procedures which a given module makes available for use within other program units or modules. Aside from the fact that the keyword IMPORTS is replaced by EXPORTS, it has exactly the same form as an IMPORTS item, and is subject to the same restrictions.

Note that no procedure can be EXPORTED from more than one module. On the other hand, a procedure defined within a module M1 but neither exported nor imported by it will be local to the module and can very well have the same name as a different procedure defined in another module M2, even if M2 exports (but M1 does not import) this procedure.

An access specification occurring in a module, program, or library has the same form, and is subject to the same restrictions, as an access specification in a library. Such an access specification is used to document the global variable accesses made and the procedures exported and imported by the module program, or library in which it occurs, and is used only for documentation, so that if it occurs at all it should be identical with the access specification supplied for the same module or program in the DIRECTORY which precedes it. Since libraries can neither access program-global variables nor IMPORT procedures from a module or program, an access specification in a library must consist of a single EXPORTS item only. As with modules, a procedure P defined in a library but not exported by it is local to the library and; can have the same name as a different procedure defined in some other library, program, or module.

Note that the libraries imported by a module or program, or by another library, are not listed in the directory which precedes them. Instead, they are listed in a library item within the importing module, program, or library. A MODULE, PROGRAM or LIBRARY L1 which lists another library L2 automatically imports all the procedures and functions which L2 exports.

A concluding note concerning use of these facilities. In subdividing large programs into modules and libraries, one's main aim will be to subdivide the full collection of procedures which constitute it (possibly amounting to many hundreds of procedures altogether) into sensible chapters, each containing procedures which are relatively tightly coupled to each other but which are only loosely coupled to procedures placed in other

modules or libraries. Close couplings will develop if procedures share variables globally, or when one procedure makes detailed assumptions about many of the data structures used by another. Procedures should be structured, and partitioned among modules and libraries, in ways calculated to avoid these couplings, and to minimize them effectively when they are unavoidable. It is particularly important to avoid accumulation of large numbers of shared global variables at either the MODULE or the DIRECTORY (i.e. program-global) level.

Ideally, a MODULE should consist of no more than a few dozen procedures, and should be considered a candidate for further subdivision when this informal limit is exceeded.

An extended example showing the use of SETL's larger program structuring facilities is found in Section XXX of Chapter XI.

9.2 Separate compilation and 'binding' of program subsections.

When long SETL programs (i.e. programs more than a few thousand lines long) are being developed, the time required for compilation becomes significant. To have to spend much time recompiling long programs after just a few of their lines have been changed is annoying, and to obviate this annoyance the SETL system allows the modules and libraries of a large program to be compiled separately. Precompiled forms of such modules and libraries (called 'Q1' or 'intermediate code' files) can then be saved, and combined or 'bound' with other subsequently compiled program sections, to produce a final, executable, SETL program. Moreover, the SETL compiler can be used to combine several seperately produced intermediate code files into one single file of the same format, thereby saving part of the expense of repeated intermediate code file binding.

The form of intermediate code saved for subsequent 'binding' is exactly the form of code produced as output by the second (semantic analysis) phase of the SETL compiler (as we have noted, this output is caled 'Q1' text.) To save this output for subsequent binding, you must either

(a) Halt the compilation process immediately after the semantic analysis phase (see Appendix 8.5.1.2 for an explanation of how to do this.)

or

(b) Prevent the third compiler phase (code generation phase) from erasing the intermediate (Q1) file passed to it (as it would normally do at the end of code generation.) To prevent this erasure, the control-statement option SIF (save intermediate file; see Section 8.5.1.2 for additional explanation) must be set.

When several seperately compiled modules and libraries, all available in the 'Q1' format, are being bound together, they are first read in order, and digested by the SETL compiler's second (semantic analysis) phase. After this, the semantic analysis phase reads any additional files representing source code newly parsed by the first compiler phase (the parse phase.) All these files are then combined, and a single composite Q1 format file representing all this input analysis is output by the semantic phase. This output can itself be saved, and combined (at a later time) with still other Q1-format files and with fresh parse output, to produce a still longer Q1 file. Alternatively, a Q1-format file representing a complete (complex or simple, see Section 9.1) SETL program can be passed to the compiler's code generation phase, to be turned into (interpretable or true machine) code and then executed. The following figure shows the main inputs to and outputs from the compiler's semantic analysis phase when it is used in the manner just outlined to 'bind' seperately compiled modules together.

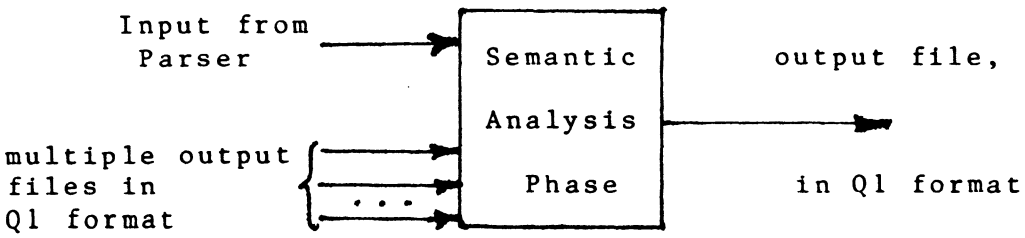


Figure 9.1: Inputs and Outputs of the SETL Compiler when the BIND option is used.

A file in Q1 format always represents a (parsed and semantically analyzed) sequence of SETL source modules and libraries (possibly including a main program module, cf. Section 4.1 following 4.1.1), in some specific order, and could be produced simply by arranging this source code in appropriate order, and compiling it. The rule determining the logical order of modules in a Q1-format file produced by binding is explained below.

The inputs to the compiler's semantic analysis phase are as follows:

(i) Two files, called POL and XPOL, which are passed from the first (parse) phase of the compiler to the semantic analysis phase. Together, these two files represent a SETL source text in parsed form, ready for semantic analysis.

(ii) An additional file, called BIND. This is a Q1-format file representing precompiled modules that are to be combined with the newly parsed material represented by the POL and XPOL files.

(iii) If necessary, a third file, called IBIND. If supplied, the IBIND file is simply a list of file names (which should have whatever file name format is required by the operating system under which you are running.) If an IBIND file is supplied, the files named in it (each of which must be a Q1-format file) are read one after another by the semantic analysis phase, and combined with the POL/XPOL material (i) and the BIND file (ii), to produce one composite Q1-format file as output.

The Q1-format file produced as output by the semantic analysis phase can be regarded as the parsed, semantically analyzed form of a certain SETL source text. This text is exactly what would be obtained by concatenating the following subtexts, in order:

(a) First, the source text corresponding to the BIND file;

(b) Next, the various source texts corresponding to the successive Q1-format files mentioned in the IBIND file;

(c) Finally, the source text represented by the POL and XPOL files.

Suppose, for example, that we are working with the complex-program 'logic_analyzer' whose structure is shown in the preceding section. This consists of the following principal subdivisions:

```
(1)     DIRECTORY logic_analyzer;
        ...
        END DIRECTORY,

        PROGRAM logic_analyzer -logic_main;
        ...
        END PROGRAM;

        MODULE logic_analyzer - propositional_calculus;
        ...
        END MODULE;

        MODULE logic_analyzer - predicate_calculus;
        ...
        END MODULE;
```

We could proceed in the following way, via a sequence of seperate compilation steps, to produce a version of this program ready for execution:

(i) First, the DIRECTORY can be compiled, and saved in Q1 format, let us say in a file called DIRECT.Q1 (here, and in the next few paragraphs, we use file-naming conventions; appropriate to the DEC VAX VM operating system.)

(ii) Next, the PROGRAM and the propositional_calculus MODULE could be compiled (seperately) and the results of these two compilations stored as two Q1-format files named MAIN.Q1 and PROPOS.Q1.

(iii) Finally, the predicate_calculus MODULE can be compiled, and combined with the precompiled material (i) and (iii). To do this, the final compilation could have the source text of the predicate_calculus MODULE as its SETL source input, and in addition have the file DIRECT.Q1 as its BIND parameter (see Section 8.5.1.2). The IBIND parameter should then be a file (possible called XTRAQ1.LIS) of file names, which should contain just the following two lines:

```
MAIN.Q1
PROPOS.Q1
```

As soon as the semantic analysis phase has finished processing its input, Q1-format output representing the parsed, semantically analyzed form of the source text (1) will result.

In using the 'binding' mechanisms that have just been explained, the following facts should be noted:

(a) Either of the BIND and the IBIND parameters can be omitted, in which case no attempt will be made to read, or to bind in, the corresponding Q1 files.

(b) If only binding of previously compiled Q1-format files is desired, POL and XPOL files produced by parsing an empty SETL input file can be passed, along with appropriate BIND and IBIND parameter files, to the semantic analysis phase of the compiler for binding.

9.3 More on interpreters: the SETL machine.

The notion of interpreter appears repeatedly in these pages (see Sections 3.5,5.4.2,and 5.4.3). Interpreters are an important, indeed fundamental class of program. This is because we can regard any interpreter as a program that 'understands' a kind of simple 'language', i.e. recognizes and executes a specific set of instructions. For example, the Turtle Interpreter of Chapter 3 recognizes the instructions: FORWARD n, LEFT, RIGHT, and so on, and executes an appropriate action for each such instruction. Imperative instruction formats of this kind can be used to manage many programming tasks.

Structuring a program as an interpreter has a number of advantages :

a) An interpreter has a simple modular structure. Even though the instructions it handles should always reflect some unifying intent, each such instruction can be programmed independently of the others.

b) As a consequence of a), an interpreter is easily extensible: it is generally easy to add new instructions and to program the actions corresponding to them without affecting existing portions of the interpreter.

c) Since any interpreter defines a kind of 'language', we can extend the family of programming languages at our disposal by writing interpreters. Once we have implemented a language by writing an interpreter for it, we can solve further programming tasks by writing programs in the new language. Often this is the most effective way of attacking a problem: invent a language L in which the problem is eaily solved, program a solution to the problem in L, and then write an interpreter for L.

In what sense can we speak of interpreters as defining new programming languages? Any programming language has two separable aspects, namely its 'syntax' and its 'semantics'. These important terms can be defined approximately as follows:

i) Syntax: The syntax of the language is the externally visible grammatical structure in which valid statements of the language must be couched. (The syntax of any language can be described by means of a formal syntactic notation like the syntax graphs that we have used to describe SETL.)

ii) Semantics: The more elusive notion of semantics includes all those aspects of the language that determine the operational meaning of its primitve constructs , i.e. the result of the actions that execution of each construct requires.

Note in this connection that different languages may give different syntactic forms to statements that have very similar meanings. For example, the action of incrementing a counter variable by one is written as follows in five well-known programming languages:

```
I = I + 1                ( in FORTRAN)
I = I + 1;               ( in PL/I)
I := I + 1;              ( in PASCAL)
```

```
I ++;                    ( in C )
I +:= 1;                 ( in SETL)
```

It is clear that in spite of their different syntactic forms, all these statements have the same operational meaning, i.e. the same semantics. On the other hand, the command

```
FORWARD 10
```

of the Turtle language (let's call it TURLAN) has no semantic equivalent in most programming languages. Its meaning can be explained in English, as we did when first describing TURLAN, or it can explained to an informed reader by showing him the SETL code which is executed when the FORWARD command is encountered by the TURLAN interpreter. Clearly, the important aspects of this command do not lie in its syntax: something like

```
ADVANCE BY 10;
```

would do just as well as a syntactic alternative for the FORWARD statement. Regardless of how we choose to write it, the important characteristic of this construct is that its execution models the motion of an object in two-dimensional space, and that this motion may produce a visible track on a drawing. This is what constitutes the semantics of the statement, irrespective of whether it is given the name 'FORWARD' or the name 'ADVANCE'.

An interpreter defines a language in the sense that it provides a semantic specification of the meaning of each command in the language. It is sometimes said that the semantics of a language L is best defined by giving the rules of action some 'abstract machine' M which understands L and executes the actions specified by elementary 'commands' of L. From this point of view, an interpreter I for L is just an implementation of the abstract machine M, i.e. it is a program (written in some other language OL) that behaves the way M should. Of course many interpreters can be written for the same L, and these intepreters can differ in programming details, in the language in which they are written, etc. However, the visible behaviour of all these interpreters (which is to say their output for a given input sequence) is identical. All these interpreters are logically equivalent implementations of the same abstract machine M.

Given two general-purpose programming languages L and OL of comparable power, it is always possible to write an interpreter for either of them using the other. It is even possible to write an interpreter for L in L itself (for example, an interpreter for SETL in SETL). This is often often done to provide a self-consistent definition for a new language.

In the following pages, we will proceed to sketch such an interpreter for SETL. In doing this, we have two goals in mind:

a) This exercise will give us more insight into the general nature of interpreters and introduce the important notion of intermediate language and of 'interpretable code'.

b) The interpreter to be sketched will illustrate parts of the structure of

the actual implementation of SETL. This will advance our understanding of efficiency and data structure considerations that we have neglected thus far, and will prepare us for a subsequent discussion of the data-representation sublanguage of SETL.

9.3.1 An interpreter for SETL.

If we compare our simple Turtle language with SETL, it is clear that a full compiler for SETL must be a considerably harder program to design. The reason is not just that SETL is a much bigger language, but also that the grammatical structure of the two languages is very different. The grammar of TURLAN is very simple: every sentence is a command with one or zero arguments. The grammar of SETL is much richer, and a SETL statement is by no means a rudimentary 'command'. For example, if we examine a simple statement such as

```
(1)    squares := { [i, i ** 2]:i IN [ 1,3..21] };
```

we notice that it specifies a whole series of actions:iteration, tuple forming, set forming, assignment, etc. The following expanded SETL fragment gives a more detailed account of these actions:

```
(2)         squares := {};

            (FOR i IN [1,3..21])
                    pair := [i, i ** 2];
                    squares with:= pair;
            end FOR;
```

The fragment (2) has the same net effect as (1):after its execution, the variable -squares- is a map whose range is the set of squares of the first 11 odd numbers. However, the statements of (2) are much simpler than the single statement (1). Each step of (2) describes a relatively simple action (assign the empty set to a variable, add an element to a set, etc.) Nevertheless, the fragment (2) is not yet simple enough to be handled by a program as straightforward as our TURLAN interpreter. To simplify further, we need to expand version (2) into something like the following:

```
(3)                 squares := {};
                    i := 1;

            LOOP:   IF i > 21 GO TO out;

                    i2 := i ** 2;
                    pair := [];
                    pair with:= i;
                    pair with:= i2;
                    squares with := pair;
                    i := i + 2;
                    GO TO loop;
            out :
```

The fully expanded code (3), with its labels, jumps, individual tuple insertions and so on, is much less readable than our original one-liner. But it constitutes a list of rudimentary actions, each one of which is

simple enough to be handled by an interpreter. To simplify one final step further, it is useful to write the commads appearing in (3) using a more stereotyped syntax than that of SETL. This leads us to introduce an 'intermediate language' whose statements have the same semantic meaning as the SETL statements of (3), but whose syntactic structure is more uniform and hence easier to process.

Specifically we will represent command in (3) as sequence of four components :

operation first operand second operand result (optional)

Such an sequence is customarily called a quadruple, for obvious reasons. Instructions written in this way are also known as 3-address code, because each instruction names (up to) three quantities: two operand(s), and a result.

Written in this way the five lines that follow the label 'loop' in version (3) would take the following form:

(4)
```
expon   i         2     i2
assign  nulltup   -     pair
with    pair      i     pair
with    pair      i2    pair
with    squares   pair  squares
```

Note that code of this kind restricts us to use no more than two arguments in each quadruple, which forces us to break up complex expressions into sequences of simpler steps. For example, in reducing (2) to (3) we introduced a whole assignment instruction just to calculate i**2. In (2), the expression i**2 had no 'name':it just appeared as a component of a tuple. In version (3) we isolate the calculation of i**2 into its own instruction, and give the resulting value a name in order to use it at a subsequent instruction. Such a name, which does not appear in the original program (1) but is generated when (1) is translated into a syntactically simplified form, is known as a 'temporary variable'. One of the ongoing activities of a 'compiler', which translates syntactically complex programs like (1) into intermediate quadruple forms like (3), is to generate such names whenever they are needed to simplify complex expressions.

It should come as no surprise that in the real SETL system SETL source programs like (1) are actually converted into an intermediate quadruple form like (4). This is done by the SETL compiler, whose output is precisely a sequence of quadruples. The details of this translation need not concern us here. What is relevant to our discussion is the fact that the run-time system of SETL, which is what actually executes all your programs once they have been compiled, is an interpreter for this intermediate language of quadruples. This SETL run-time interpreter has a repertoire of about 250 operations. Most interpreters for complex languages work in this way: the language is first translated into a simpler form, consisting of a small set of command-like statements, and then this intermediate representation is processed by the interpreter.

The full intermediate code representation of our initial example (1) is as follows:

```
assign   nullset    -      squares
assign   1          -      i
ifgt     i          21     10
expon    i          2      i2
assign   nulltup    -      pair
with     pair       i      pair
with     pair       i2     pair
with     squares    pair   squares
plus     i          2      i
go       -          -      3
```

Note that in this intermediate representation, the labels introduced in version (3) have been replaced with numbers . For example, the intermediate code instruction

```
go       -          -      3
```

instructs the interpreter to proceed to the 3rd instruction in the sequence. Similarly, the conditional instruction , which is instruction number 3 in the above sequence, namely

```
ifgt     i          21     10
```

directs the interpreter to proceed to the 10th instruction in the sequence if the condition (i > 21) is met. If this condition is not met, then the interpreter simply continues on from instruction 3 to instruction 4.

To summarise: the intermediate code version of a SETL program is a sequence of quadruples representing the original program. Some quadruples trigger computations while others affect the order of execution, i.e. affect the flow of control through the sequence of quadruples.

The foregoing explanation puts us in position to sketch the overall structure of the SETL interpreter. As usual, the main component of the interpreter is a CASE statement, each of whose tags corresponds to an instruction in the intermediate language. This CASE statement is executed within a loop, (the 'main interpretive loop'), and each step through the loop performs the following actions:

(a) Fetch the next quadruple to be executed.

(b) Unpack the quadruple into instruction, arguments, and name of

result.

(c) Execute the code corresponding to the current instruction. This generally involves fetching the values of the arguments, performing some calculation, and assigning the output of this calculation to the result parameter of the quadruple. In the code that follows, the relationship between a variable and its current value is represented by means of a map called VALUE, whose domain is the set of identifiers present in the program

being interpreted.

(d) Finally, if the instruction is not a STOP, we go back to (a).

Although highly abbreviated, the following SETL code indicates the main features of this structure:

```
PROGRAM interpreter;

read(code);                          $ The tuple of instructions to be
                                     $ interpreted.

next := 1;                           $ Index for next quadruple to execute.

LOOP DO                              $ Main interpreter loop.

    [opr, arg1, arg2, res] := code(next); $ 'unpack' the quadruple
    next +:=1;  $ advance (provisionally) to the next instruction

    CASE opr OF

        ('assign'):     value(res) := value(arg1);

        ('plus'):       value(res) := value(arg1) + value(arg2);

        ....                                $ code for other instructions.
        ....                                $ would come here

        ('go'):         next := res;    $ This operation modifies the
                                        $ 'program counter'  -next-
        ....
        ....

    END CASE;

END LOOP;

END PROGRAM;
```

The variable -next- is usually called 'the program counter' or the 'instruction counter'.

After each instruction, the interpreter determines the next instruction to execute. After a computational instruction, addition or assignment, the next instruction is simply the next instruction in the sequence, as is reflected in the statement :

```
next +:= 1;
```

Any instruction, such as a 'go' instruction; that affects the flow of control will reset the -next- indicator to some other value, as is shown in the case of the 'go' instruction above.

9.3.2 Memory management and data-structures.

The previous section evades several important questions that we must now try to answer. Writing an interpreter for SETL in SETL may be a reasonable way of describing the nature of interpreters, but it can't possibly be the way SETL is implemented, because something would have to to execute the interpreter program itself ! In fact, the real SETL interpreter is not written in SETL, but in a simpler, lower-level language, which in turn is translated into (you have guessed it) some intermediate representation which...But there is no infinite regress here: the final instructions produced by this sequence of translations can be executed by the computer 'hardware' itself, which is to say that the ability to execute these most elementary instructions is 'wired' into the physical structure of the machine on which your program runs.

Now the instructions that can be 'wired in' to actual hardware, that is to say the instructions that a computing machine can execute directly, are always of a much simpler nature than the instructions of our hypothetical SETL machine.

One of the things that simplifies programming in SETL is the fact that the language makes available such complex structured objects as maps, tuples and sets. This frees the SETL user of all concern with with the physical location of these objects in the machine, and with the ways in which they are retrieved, modified, created, deleted, etc.. But the actual computer hardware does not have the built-in capabilities to deal with these structures, i.e. the bare computer knows nothing about sets, maps, tuples, membership tests and the like. In creating the SETL interpreter it is therefore necessary to program the manipulation of these objects explicitly. To understand the efficiency consequences of this fact, we must now examine the way in which SETL objects like tuples, sets and maps, are actually represented in memory. In order to do this, we must first say something about the capabilities typically available in a bare machine, which is to say the repertoire of instructions supported by the 'hardware' itself.

Machine level operations.

Physically, a computer consists of two linked subparts :

a) A memory, within which data can be stored.

b) A processing unit which reads items from the memory, combines and modifies them in various ways, and stores the results back in memory.

The memory consists of a number of storage units, called words. We can think of the memory as a tuple of some fixed size. Each word has an 'address', which is simply a positive integer, which when used as an index allows us to refer to that specific word.

In turn, each word in memory consists of a fixed number of information units called bits. A bit is a binary digit, i.e. it is representable by a 0 or a 1, so that we can think of a machine word as a tuple of zeroes and ones. Word size and memory size vary from computer to computer. Typical values

are as follows:

memory sizes range from 4000 words to several million words.

memory words are typically 16, 32, 36, 48, 60, or 64 bits long.

In this section we will use -W- to refer to some typical word size. The fact that is central to the following discussion is that the operations that a given machine can execute directly (usually called its 'instruction set') are almost without exception instructions that involve one or two machine words of input, and yield one machine word of output. Anything more complex must be programmed as a sequence of instructions. Let us now briefly examine briefly the capabilities of a typical instruction set. To describe this, we will make use of the following notation:M designates the machine memory, which we regard as a tuple of words, and M(i) designates the contents of the ith word of memory. The contents of a machine word can be regarded as a sequence of bits, as we described above. However, it is also possible for the hardware to regard the contents of a machine word as the binary representation of an integer, or some encoding of a floating point quantity, or as one of more alphabetic characters. In other words the machine has no concept of data object 'type'. Each instruction implicitly determines which interpretation the machine will give to the contents of the memory locations that it references.

Bearing this in mind, we can classify machine instructions into the following classes:

(i) Transfer instructions.

These instructions transfer the contents of one location to another, i.e. perform operations like

M(j) := M(i)

In some cases, it is possible for one machine instruction to cause the transfer of several memory words, or the transfer of a portion of a word, i.e. of a few bits, from one location to another.

(ii) Arithmetic instructions.

All sizeable machines supply the four basic arithmetic operations on integer quantities: add,subtract, multiply, divide. When these operations are executed, the contents of words in memory are interpreted as binary representations of integers. Usually one bit of a word representing an integer is reserved to represent the sign of the integers. If the word has size W, there are therefore W-1 bits available to specify the size of an integer, which allows the representation of numbers whose magnitude is in the range 0 thru 2**(W - 1)-1 .

(iii) Bit-manipulation instructions.

The instruction set of most machines includes operations that regard the content of memory words as a sequence of boolean quantities, i.e. regard each one-bit as an encoding of TRUE and each zero as FALSE. Instructions in this class perform boolean operations (AND, OR, NOT) on these

representations. These instructions are performed in a single step on all the bits of a given machine word. Typical instructions in this group include the following :

1. Negation:replace each bit in a machine word by its negation, i.e. replace each zero by a one, and viceversa.

2. AND:form the bitwise AND of the contents of two machine words, and place it in a third. In other words, the ith bit of the result is obtained by ANDind the ith bits of the two operands.

3. Bit OR: Similarly, perform a bitwise OR operation on the contents of two words, and place the result in a third.

(iv) Indexing instructions.

Given that the address of a machine word M(i) is simply an integer, this address can itself be stored in some other machine word, say M(j), and subsequently used to retrieve the contents of M(i). In this case, as in the case of arithmetic operations, the machine interprets the contents of M(j) as an integer, and performs an instruction like

$$M(k) := M(M(j))$$

That is, a typical indexing instruction takes the contents of the word whose address is contained in M(j), and transfers these contents to M(k) .

(v) Test and branch instructions.

Every instruction set provides various test operations whose outcome determines the instruction which the processing unit will execute next. Instructions of this type are the machine-level equivalent of IF statements in SETL. Among others, the following conditions can usally be tested for:

a) Test for zero:this condition is TRUE if all the bits of a given machine word are zero.

b) Test for equality:this condition is TRUE if the contents of two machine words are bitwise equal.

c) Arithmetic comparisons:interpret the contents of two machine words as integer quantities, and compare these quantities for (equality, greater than, less than, unequality, are commonly conditions that can be tested).

Modern instruction sets contain hundred of individual instructions, and the list above is a small but representative sample of what is commonly available. Regardless of the specific details of a machine's instruction set, what is important is that all these insructions manipulate one or two words of output and produce a single word of input. Moreover, these insructions can be executed at very high speed. Typical modern computers perform between 1 million and 20 million cycles per second. A transfer operation, an addition, or a bitwise AND take one machine cycle, a multiplication may take 3-5 machine cycles.

In chapter 10, we will discuss the implementation of SETL primitives, that is to say the way SETL structures and operations are represented at the machine level. Once we understand the ways in which tuples, sets and maps are represented and manipulated, we will be in a position to discuss questions of efficiency, and to describe ways in which the execution speed of SETL programs can be improved by choosing data representations appropriately.

9.4 Appendix. A machine interpreter in SETL.

In the previous section, we have outlined the organization of the SETL interpreter, and this led us to descend from the rich set of primitives of the SETL language, to consideration of the restricted capabilities of typical hardware. In this discusiion, we treated machine operations as primitive actions and did not attempt to analyze them or decompose them into simpler elements. It should nevertheless be clear from what we have said that each machine instruction, involving as it does a few machine words, manipulates and modifies a few dozen bits of information. Thus, a description of these actions in terms of bit-level operations will give us a yet closer view of what the hardware actually does. It is interesting to note that in many modern machines, the 'machine' instructions discussed above are not indivisible actions but are actually performed as a sequence of still simpler steps, and that these simpler steps are themselves instructions for an interpreter (once again !) . This interpreter (called the microcode interpreter for the machine) is the one which is actually realized 'in the wires' i.e. this interpreter is actually built as a series of gates and transistors on a chip. Thus the notion of interpreter permeates the subject of language and machine implementation from top to bottom, A working system consists a series of languages and interpreters, each language being interpreted by an interpreter written in the language below. For example when you execute a TURLAN program, the following is actually taking place :

```
TURLAN program (LEFT, RIGHT, FORWARD, etc.)
                runs on
TURLAN interpreter , written in SETL,
                runs on
SETL interpreter , written in LITTLE.(A low-level language)
        which was translated into a machine program,
                runs on
        computer (VAX/780 for example) whose instruction set is
                executed by
        microcode interpreter(a physical device)
```

How shall we describe the nature of the microcode itself ? We do not have any simpler 'system' in terms of which to describe it, but in fact there is no conceptual problem in describing it in SETL ! This may appear at first sight a bizarre endeavour, but there is no contradiction: recall that given two sufficiently rich languages, it is always possible to write an interpreter for either, using the other. Use of SETL to describe the bit-level structure of machine operations gives us a detailed semantic specification of machine language, which has the advantage of being written in a language with which we are now thoroughly familiar.

To begin: we can describe each machine word in SETL as a tuple of zeros and ones This allows us to refer to individual bits. we will regard these bits as arithmetic quantities on which addition, etc. can be performed. W1,W2,W3 will refer to the memory words involved in an operation. Code fragments representing the various machine operations are as follows:

a) transfer operation.

```
W2(1..W) := W1(1..W)
```

b) Bit operations.

b1 negation:
```
$ W2 := not W1

W2 := [ (bit + 1) mod 2:bit in W1];
```

b2. AND:
```
$ W3 := W1 AND W2

W3 := [ W1(i) MIN W2(i):i in [1..W] ];
```

b3. OR:
```
$ W3 := W1 OR W2

W3 := [ W1(i) MAX W2(i):i in [1..W] ];
```

c) Arithmetic operations.

--------TO BE CONTINUED---------

9.5 Exercises (TO BE ADDED)

$

CHAPTER 10

THE DATA REPRESENTATION SUBLANGUAGE

The 'level' of a programming language is determined by the power of the semantic primitives which it provides. The operations provided by the ordinary low-level languages, e.g. languages of the FORTRAN type, all lie close to those elementary operations with a few dozen bits of input and output which computer hardware implements directly. Languages of somewhat higher level, e.g. PL/I, PASCAL, or ADA, supplement these primitives with more advanced pointer-oriented memory management mechanisms and also support recursion; nevertheless, even these languges stay close to operations which can be translated into efficient machine code in relatively obvious ways. SETL aims more radically than any of these languages at simplification of the programmer's task, for which reason it supports use of abstract objects (sets and maps) whose best machine-level representation is not obvious. Of course, many possible representations for objects of this kind are known, but which representation is best will vary from program to program in subtle ways that depend on the specific operations which a program applies to the objects which it manipulates. If the most effective representation of a program's data objects is not chosen, efficiency will suffer, and it is this efficiency barrier that has prevented rapid and widespread adoption of very high level languages like SETL.

If efficiency is an important enough consideration to justify the effort involved, a SETL program can be translated manually into a more efficient version written in a lower-level language such as PL/1, PASCAL, or Ada. A programmer using this approach will soon notice that many (but not all) of the efficiency-enhancing changes made during translation of an original SETL program are stereotyped in character and serve only to make use of advantageous data structures. The SETL facility to be explained in this chapter, namely its data representation sublanguage, aims to make it possible to attain efficiency without laborious translation becoming necessary, simply by declaring what data structures (chosen from a library of such structures) are to be used to represent each of the logical objects appearing in a program. Then elaboration of more efficient code sequences can be left to the SETL compiler. Programming in this style, which begins with a program in which algorithmic actions are represented but data structures are ignored, but then subsequently goes on to choose efficiency-enhancing data structures, exemplifies the important general idea of programming by successive refinement of an original program text.

SETL's representation sublanguage adds a system of declarations to the core language described in the preceding sections (which for emphasis we will sometimes call 'pure' SETL). These declarations control the data structures that will be used to implement an algorithm that has already been written in pure SETL. Ideally no rewriting of the algorithm should be necessary. A pure SETL program to which data structure declarations have been added is called a supplemented program. In the absence of error a supplemented program SP must always yield the same result as the pure program PP that it incorporates. (However, if errors or inconsistencies are present, then SP and PP are allowed to abort differently; and certain inconsistencies, detected in SP but not in PP, can cause SP to abort even if PP does not).

Chapter Table of Contents

10.1 Implementation of the SETL Primitives

To implement SETL, all its data objects must somehow be represented by sequences of machine-level memory words, and all its primitive operations must be represented using sequences of the high-speed but very elementary machine-level operations described in Section 9.3.2. We shall now outline the way in which this is done. To do so, it will be convenient to represent data layouts in machine memory diagrammatically. As noted in Section 9.3.2, the memory of a computer can be thought of as an array M of words, each able to store a fixed number W of binary bits (zeroes and ones). Such patterns of bits can be interpreted as encodings of integers, and hence can be used, when desired, as the indices of other elements of M.

We will picture subareas of the memory array M as sequences of rectangular boxes or 'memory cells', each holding a word. If one memory cell holds a value M(i) which, regarded as an integer j, is the index of the memory cell M(j), then we will sometimes regard M(i) as holding a 'pointer' to M(j), and draw an arrow from the box representing M(i) to the box representing M(j), as in the following figure :

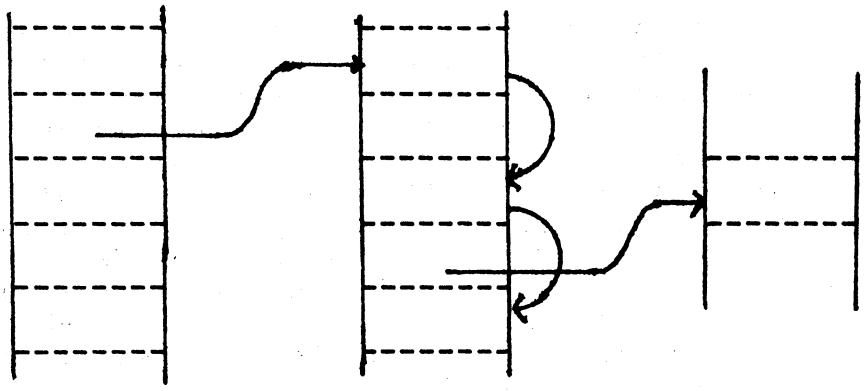


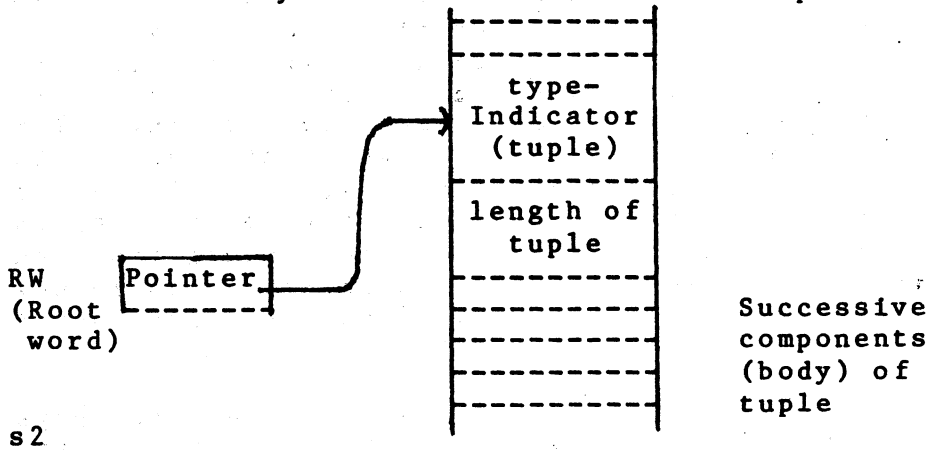
Fig. 10.1 Sections of memory, showing cells which store the indices of other cells

Where convenient, we will label the picture of a memory cell with an indication of its contents. Note that inter-cell 'pointers' like those in Fig. 1 can be followed at very high speed by using the machine-level operation (b) described in Section 9.3.2.

If the data representation language to be described later in this chapter is not employed, a narrow range of highly standardised data structures will be used to represent SETL data objects. The most significant representations are those of sets, maps, and tuples. Since tuples are simplest, we shall describe their representation first.

The standard representation for tuples.

As for all other SETL data objects, the representation of a SETL tuple begins with a single memory word, RW, called the root word of the tuple. However, since the group of W zeroes and ones which a single machine word can hold are by no means sufficient to represent the



s2
Fig 10.2 Machine-level representation of a tuple.

(possibly very long) sequence of components of the tuple, this root word simply points to another location in memory, at which the actual representation of the tuple is located (see Fig. 2). This representation begins with a tuple 'header word' which tags the information which follows as a tuple. Next comes a word containing the length of the tuple, after

which there follows a succession of root words representing the successive components of the tuple. Note that this representation makes it easy and fast to retrieve the i-th component of a given tuple t. Aside from complications caused by error cases, which arise if i exceeds the length of t (or is negative, or is not an integer, etc.) all we have to do is take the integer value contained in the root word RW, add (i+2) to it as an offset, and retrieve the word to which this sum points. How expensive is a tuple retrieval operation? The mechanism we have just outlined takes a few (less than 10) machine instructions. However we also incur another cost when we evaluate the primitive SETL operation A(i), namely since we must check the types of both A and i. More specifically, the following tests must be performed before the ith element of x is retrieved :

i) Determine the type of A. A could be a tuple, a string, a map , or could have some other type (for which the operation A(i) might be invalid).

ii) Determine the type of i. If A is a tuple, then i must be an integer, or the operation A(i) is invalid.

iii) Compare the value of i with the length of A. If i > # A then A(i) is OM.

These various tests also require a few dozen machine instructions, and therefore add a substantial overhead to the cost of the indexing operation.

10.2 The standard representation of sets

The machine representation of tuples is straightforward and relatively problem-free: a tuple, being an ordered (described above) sequence of components, can be stored as an ordered sequence of words in memory. When we access a tuple to obtain or modify one of its components, we simply use the index of the desired component to address the component.

Sets are manipulated in a different manner. To see why this is advantageous, consider the basic membership operation such as (x IN s), which asks whether the current value of x is to be found among the elements of s. Determining this logically involves a search of the elements of S. Searching is also required to implement other basic set operations. For example, when we compute the expression (s WITH x) we first search s to ascertain that the value of x is not already contained in s, and only if it is not do we perform the insertion operation. In contrast to operations on tuples, which always access components using their position, operations on sets need to locate elements whose value, rather than position, is known. For this reason, sets and maps are often called 'content addressable structures'.

Before going on to describe how SETL sets are actually stored, it pays to consider one obvious, though in fact not ideally effective, representation for them: Why not store sets as tuples? The only objection to this choice is one of efficiency. Consider again a membership test : (x IN s) . If the elements of s were stored sequentially in some arbitrary order in memory, we would have to compare each one of these elements with x to determine the truth value of the membership predicate. If the cardinality of S is N, then in the worst case it would take case N comparisons to compute this predicate, making this an expensive operation if

N is large. Since the membership operation is basic to all other set primitives (insertion, deletion, union, intersection, map retrieval and assignment) an efficient membership operation is indispensable to an efficient implementation of sets, and therefore this obvious approach is unacceptably inefficient (for large if not for small sets).

The key to a better representation for sets is the following observation: sets have no a priori order, so that their elements can be stored in any convenient fashion. This suggests that we choose an organization which makes it easy to retrieve an element, given its value. To begin to see how this might be done, suppose first that S is a set of alphabetic strings. Then a fairly obvious idea is to store these strings in alphabetic order, in a contiguous sequence of memory locations, and regard this sequence as the representation of the set S. This would speed up membership tests because we could then perform a binary search (see Section 4.4.3) to determine where in the set a given string was. Further improvement in performance can be obtained if we keep track of the location at which strings with a given first character begin. (Very much like the thumbing marks in a dictionary). This would further restrict the range over which we had to search. The actual SETL representation of sets pushes this idea still further, using a data structure called a 'hash table' which allows the VALUE of a given object x, to be mapped to a small range of L locations in which x might be found. In order to apply this technique to sets of elements of arbitrary kinds, we must be able to construct such mappings for objects x of any type. The result of applying such a mapping to x must be a single location, or a very small range of locations, at which the element x will be found if it is present at all. In addition, the data structure we use must allow insertions and deletions to be made easily: note that this is not the case for the alphabetic ordering just suggested. The kind of mappings from values to locations that we will use is called a called hashing function, which is why the structure that is organized by means of a hashing function is called a hash table.

To explain how this data structure works, it is convenient to consider an example, and for specificity's sake we will explain the internal representation of a set of integers q. The trick involved in 'hashing' is to use q itself to determine the table address at which the set element q will be held. Any function H which converts q into a numerical index to a table of reasonable size can be used: all that is desired is that H should 'scatter' the values H(q) in reasonably even fashion over the available table addresses, thus ensuring that we do not attempt to store too many items q in (or near) the same table address. The tables which the SETL implementation uses to represent sets always have a number of entries equal to a power of two, i.e. either 4, 8, 16, 32, etc. table entries are used, depending on the size of the set being stored. The size of the table is adjusted to the size of the set, so that if a set s grows by the addition of new elements, it will eventually be moved to a larger table, and if its shrinks substantially because elements are being removed from it, it will be moved to a smaller table. In this way the SETL implementation ensures that at least half the available entries in the table used to represent a set are occupied, and that table 'overloading' (explained in more detail below) never rises to more than two elements per table entry.

In accordance with the preceding remarks, we will suppose that a table of size four is being used to represent the five element set (*) shown above. As stated earlier, the standard function H(q) used to map elements to their table positions can be arbitrary, but we want it to 'scatter' fairly evenly. This is to say that, given integers i1,i2.. that are to be placed in a set s, we want the values H(i1), H(i2)... to be distributed evenly over the range table indices, i.e. 1 to 4. Any kind of arithmetic function that yields a number in this range is acceptable as a hashing function. Typically H is some otherwise meaningless sequence of operations, chosen for its simplicity, and for the eveness with which values H(x) will scatter. For example, something like the following might be used:

```
H(q)=((q+112)** 2 DIV 99 MOD 4) + 1.
```

(Here we are being suggestive rather than precise; optimal choice of 'hash functions' like H is a matter that has been studied very extensively, and we do not wish to say that precisely this function is used in the SETL implementation, but only to show something of how a hashing technique works).

Note that by reducing the quantity (q+112)**2 DIV 99 modulo 4, we ensure that H(q) always returns a value between 1 and 4, i.e. a number that can be used as an index to an entry in a table of size 4. The exact values that H takes on for the five elements of our set are as follows:

Element q :	3	17	201	48	722
Value H(q) :	2	1	2	3	2

These H-values imply that we will store 17 in the first entry of the table representing (*), 48 in the table's third entry, and that we would want to store 3, 201, and 722 in the table's second entry. However, since each table entry can hold no more than one set member, we are forced to place two of these three elements elsewhere. What is done is to place them in separate locations, but chain them into a list (called a 'clash' list) by means of pointers. The starting location for the clash list containing an element q is simply the hash-table location indexed by H(q).

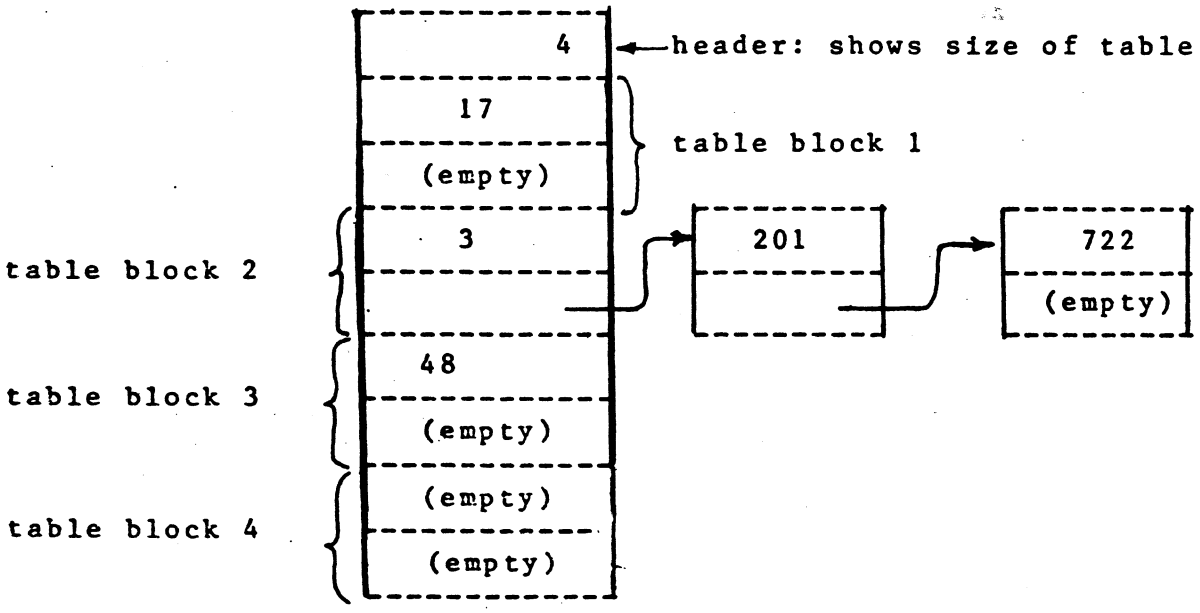


Fig. 10.3 Machine-level representation of the set {3,17,201,48,722}.

The following examples will clarify the way in which we would use the hash-table representation shown in Fig. 3. If asked to make the test (201 IN s), where s is the set shown in Figure 3, we would calculate H(201), obtaining the result 2, which tells us to examine the second two-word block of the table appearing in Figure 3. Upon examining this block, we would note that a chained list L starts in it, and would then walk down the list L, looking for the element 201. This will be found when we reach the second element of L. Similarly, if asked to make the test (33 IN s), we would calculate H(q)=1, and accordingly would examine the first block of the table. It would then be seen that the quantity 33 is not present in this block, and also that the subsidiary 'clash' list that could start in this block is empty. This relatively efficient computation would therefore tell us that the value of (33 IN s) is FALSE.

To summarize : when we insert a new element into a set, we calculate its 'hash code' in order to determine where in the hash table for S it should be stored. When we perform a membership test on S, we calculate the hash code of the element to know where in the table we must look , and WE USE THE SAME HASH FUNCTION EACH TIME.

Maps f are stored in much the same way as sets. (After all, maps are just sets of pairs). However, the hash code of a pair [x,y] is taken to be the hash code of x, that is, of the domain element of the pair. This makes it easy to find y given x, i.e. to calculate f(x) from x. Fig. 10.4 depicts the internal representation of a SETL map; note in particular that the table entries in the representation of a map are somewhat larger than those used to store elements of sets which are not maps (compare Fig. 3). We enlarge the table entries in the representation of maps in order to store range elements in immediate proximity to the domain elements to which they correspond.

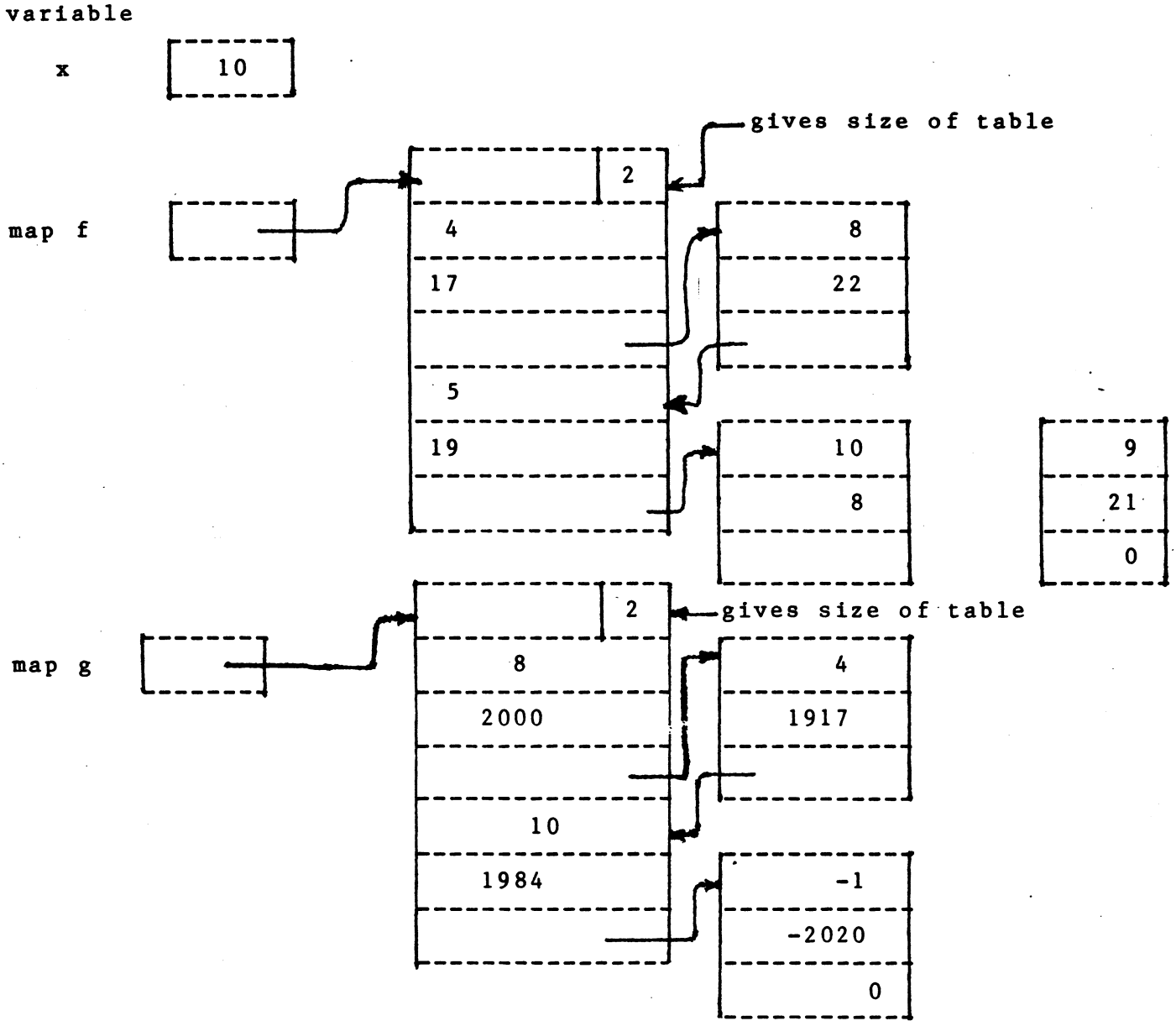


Fig. 10.4 Standard SETL representation of an (integer) domain element x, and of several maps.

In working through the last few pages, the attentive reader may have realized that more details have been concealed than revealed. How do we calculate hash indices H(q) for quantities q that are not integers? What representation is used for maps that are not single-valued? How do we iterate over sets, how do we test sets for equality? What representation is employed for a set of pairs that is not being used as a map? The SETL implementation, i.e. the SETL run-time support library, must face all these questions and provide effective solutions for them. However, to explain the goals of the data representation language to which this chapter is devoted, we need not, and shall not, describe any of these finer details. All that is important to us can be summarised as follows: To make the basic test (x IN s), or to evaluate f(x) when f is a map, we must perform the following actions :

i) Calculate the hash code of x.
ii) Find the starting location of the hash table for s (or f).
iii) Index this table with the calculated hash code.
iv) If x is not found at the position first examined, and there is a clash list starting at this position, examine the elements of this list until either x is found or until the end of the list is encountered.

It is clear that this sequence of operations is considerably more expensive than a simple tuple access. Typically, 50 to 100 machine instructions will be executed to complete a standard set membership test or map retrieval. This is not an unreasonable price to pay for the convenience of using sets and maps, but if possible we would like to be considerably more efficient. Gaining additional efficiency is the point of the data structure representation sublanguage of SETL to be presented later in this chapter.

The preceding discussion emphasises two aspects of the execution of important SETL operations such as membership tests and map retrievals which can be regarded as 'costly' :

a) Each instruction must check the type of its variables.

b) Hashing must be used to access content-addressable objects (sets and maps). These operations are considerably more expensive than simple memory references and tuple retrievals.

The data-representation sublanguage (DRSL) of SETL, which we will now proceed to describe, allows us to reduce the costs associated with these execution-time activities. This sublanguage gives us a mechanism for adding declarations to a SETL program, declarations which aid the SETL compiler to simplify and in some cases even eliminate expensive computations. The basic ideas used to achieve this are as follows:

a) In order to reduce expense a), that is to say the cost of the type-checking steps that must be performed before a primitive SETL operation is executed, the declarations of DRSL can be used to specify the types that the variables will have at execution time. The types involved here can be 'integer', 'boolean', 'array of strings', 'map from integers to strings', etc.) We shall call these declarations 'Type declarations' for obvious reasons.

b) In order to reduce the expense associated with hashing operations, we try to avoid repeated rehashing where possible. The DRSL gives us a means to replace repeated rehashing by direct indexing in many cases. The basic idea here is to 'remember' the location of an object after it has been placed in a set or map. The run-time structure that retains this information is called a base set, and the declarations that refer to base sets are called basing declarations. The detailed syntax and semantics of these basing declarations will be described in Section 10.4.

10.3 Type Declarations

We can divide the declarations of the DSRL into two categories : type declarations and basing declarations. Both of these have the same format, but they are motivated by somewhat different considerations, and basing declarations introduce some rather subtle concepts into the language, discussion of which we will postpone until the next section. In contrast, type declarations are quite straightforward: they describe the types which variables in a SETL program will have at run-time.

However, before specializing our discussion in this way, let us first examine the general syntax and usage of DSRL declarations, also called representation declarations, or REPRs for short. REPRs are optional declarations that can be added to a SETL program in order to improve its efficiency. REPRs added to a SETL program must be grouped into sequences of declarations bracketed by the keywords REPR and END. Such declarations must appear before any executable statements, and after any declarations for constants and global variables appearing in the same program, module or procedure. A main program can include a set of REPR declarations for the global variables declared in a VAR statement, and each procedure can have REPR declarations for its local variables. We emphasize again that REPR declarations are optional, and that not all variables in a program or procedure need to be declared. Sec.10.4 contains guidelines for the inclusion of REPRs in a program.

A REPR clause has the form :

```
<name list> : <mode> ;
```

where <name list> designates a list of one or more variable names (identifiers) separated by commas, and <mode> is a type name or a basing descriptor that applies to each of the variables in the list. An example is

```
REPR
        count, size, left : INTEGER ;
        here, there, elsewhere : STRING ;
END ;
```

Here the identifiers INTEGER and STRING are type names; the first REPR clause above declares that the variables -count-, -size- and -left- will have integer values wherever they appear in the portion of the program which these declarations govern. Similarly, the variables -here-, -there- and -elsewhere- must be string values wherever they are used. Note that such declarations refer to ALL occurrences of the variables that they name in the 'context' or 'scope' that they govern. We have seen that, in pure SETL, variables can receive values of different types at various points in the program. In the presence of REPRs this is no longer the case: values assigned to a variable for which a REPR is given must ALWAYS have the type that has been declared for the variable. The discipline this imposes on the writer is salutory: one can easily find different names for objects of different types, and it easier to understand the purpose of a program if the same name is used in the same way wherever it appears.

The systematic list which follows presents most of the modes that can be used in type declarations. Two examples of these 'modes', namely INTEGER and STRING, have appeared already. In general, modes can be either simple or compound. Simple modes describe primitive types, while compound modes

describe sets, tuples and maps. The simple modes allowed in the DRSL are the following:

INTEGER	mode of integers
INTEGER e1..e2	mode of integers constrained to be in the range e1 to e2. Here e1 and e2 must be elementary integer-valued expressions involving constants only. Examples and additional details are given below.
REAL	mode of real numbers
STRING	mode of SETL string quantities
ATOM	mode of SETL atoms (See Section 5.3).

The compound modes allowed by the data representation language are as follows.

GENERAL	This is the default SETL mode. Quantities declared to have this mode can be arbitrary SETL values.
*	The mode symbol '*' is simply an allowed abbreviation for 'GENERAL'.
SET(mode')	mode of sets all of whose elements are constrained to have mode mode'. Examples showing the use of this construct are given below.
SET	allowed abbreviation for SET(GENERAL).
SMAP(mode')mode''	mode of single-valued map with domain elements of mode' and range elements of mode''.
SMAP(mode')	This is simply an allowed abbreviation for SMAP(mode') GENERAL
SMAP	This is simply an allowed abbreviation for SMAP(GENERAL)GENERAL
SMAP(model,..,modek)mode''	Mode of single-valued k-parameter map (see Section 2.7.5) with domain elements having mode TUPLE(model,..,modek)(see below) and range elements of mode''.
SMAP(model,..,modek)	Abbreviation for SMAP(model,...,modek)GENERAL
MMAP{mode'}mode''	mode of (possibly) multi-valued map with domain elements of mode' and range elements

of mode''.

MMAP{mode'} Abbreviation for MMAP{mode'}GENERAL

MMAP Abbreviation for MMAP{GENERAL}GENERAL

MMAP{model,.. modek}mode'' mode of possibly multi-valued k-parameter map (See Section 2.7.5) with domain elements having mode TUPLE(model,...,modek) (see below) and range elements of mode''.

MMAP{model,...,modek}
Abbreviation for MMAP{model,...,modek}GENERAL

TUPLE(model,...,modek)
mode of tuple of known length k, whose j-th component is known to have mode modej.

TUPLE(mode') mode of tuple of unknown length, all of whose components are constrained to have mode mode'.

TUPLE This is simply an allowed abbreviation for TUPLE(GENERAL).

TUPLE(mode')(e) Mode of tuple of unknown length, but of estimated length e, all of whose components are constrained to have mode mode'. Here e must be an elementary integer-valued expression involving constants only.
Examples and additional details are given below.

PROC(model,...,modek)mode
mode of k-parameter programmed function (i.e. PROCEDURE) whose parameters have respective modes model,...,modek, and which returns a mode'' value.

PROC(model,..,modek)
This is an allowed abbreviation for PROC(model,...,modek)GENERAL. It can also be used to describe non value-returning procedures whose parameters have respective modes model,...,modek. (Typical uses for this and the immediately preceding PROC mode descriptor will be explained below).

PROC mode of a procedure unconstrained as to mode of arguments and of result value, if any.

OP(model,mode2) mode of infix operator whose two parameters have respective modes model and mode2, and which returns a mode'' value.

OP(model,mode2) Abbreviation for OP(model,mode2)GENERAL

OP(mode')mode''	mode of prefix operator, with one mode' argument, which returns a mode'' value.
OP(mode')	Abbreviation for OP(mode)mode''.

There is one more type declaration, having a rather special character. Unlike the other type declarations, it has an efficiency implication, namely it states that a variable all of whose values are integers will only take on values that are within he range of integers that can be handled directly by the hardware of the machine on which you are running. Integers of tis kind can be manipulated particularly rapidly. This special type declaration is

UNTYPED INTEGER	An 'untyped' integer is an integer represented in the standard machine-level integer format of the machine on which your SETL implementation runs. Operations involving untyped integers are particularly efficient. However, untyped integers are constrained to lie in the range of values for which the elementary arithmetic operations of the computer that you are using represent integer arithmetic correctly. See Appendix A for details concerning the integer arithmetic operations of the various machines on which SETL is implemented.

10.3.1 An example of the use of type declarations

Next we give a simple example of the use of REPRs, which we will apply to one of the prime-finding methods described in Sec.3.3.8.1.

```
PROGRAM primes;

REPR
    prime, next, limit, c : INTEGER;
    primes, candidates    : TUPLE(INTEGER);
    multiples             : SET(INTEGER);
END;

read(limit);

candidates := [3,5..limit];
primes := [2];
prime := 2;

(WHILE prime ** 2 <= limit)

    prime FROMB candidates;
    primes WITH:= prime;
    multiples := {prime ** 2};
```

```
    (FORALL c IN candidates)

        NEXT := prime * c;
        IF next > limit THEN quit;
                        ELSE multiples WITH:= next;
        END IF;
    END FORALL;

    candidates := [c IN candidates | c NOTIN multiples ];

END WHILE;

primes +:= candidates;

print(primes);

END program;
```

In this example, we have supplied type declarations for all variables in the program, including the loop variable c. We have not supplied size information for the tuples primes and candidates, because we do not know a priori the number of components that they will have. Note that the variable -limit-, which defines the range in which we want to find primes, gets its value from a -read- statement, and therefore its value is not known to the compiler, and cannot be used to declare any variable in the program. That is to say, if we had written the declarations :

```
candidates : TUPLE(INTEGER)(limit);
prime      : INTEGER 2..limit;
```

the compiler would reject them on the grounds that -limit- is not a constant.

Our next example concerns graphs. It is the well-known algorithm for determining the shortest distance from one vertex of a graph to all the other vertices.

As before (See Section 5.3) we regard a graph as consisting of a set of nodes (or vertices) and a set of edges. Each edge is represented by an ordered pair [from,to] of nodes. It is convenient to regard the set of edges as a map: given a node n, its image under this map is the set of nodes that are linked to n by one edge of the graph. In the program that follows, this map is called -successors-. It is a multi-valued map, because several nodes may be reachable from the same n by an edge. Each edge has the same (postive) assigned length. The length of each edge is represented by a map from edges to integers. The minimum distance from the start vertex to all the other edges , which is the desired output of the program, is a map from nodes to integers. The nodes themselves do not have a particular type: we can use integers to describe them, or strings, or atoms, depending on the application. In the REPRs that follow, we introduce the mode -node- and state that -node- can be any type (i.e. general). This allows us to represent program variables in terms of nodes, without having to be any more specific about what a node actually is.

The algorithm works as follows: we construct a set -reached- , whose elements are nodes whose shortest distance to -start- has been determined. Initially -reached- only contains -start- . Each step in the algorithm adds one node to the set -reached-. The node to be added next is chosen as the one whose estimated shortest distance to -start- is the smallest. We estimate the shortest distance from -start- to any node n as follows:

a) If there is an edge from start to n, the estimated shortest distance is the length of that edge.

b) When a node -new- is reached, there may be a path from-start-to-n that goes through the -new-. In that case, calculate the distance from start-to-n- along that path: it is the minimum distance to -new- plus the length of the edege from -new- to -n-. If this distance is smaller than the previous estimate of the distance to -n-, use this value as the new estimate.

```
PROGRAM shortest_paths;

REPR

        MODE node : general;

        successors: MMAP(node)set(node);      $ see comment above
        length    : SMAP(node, node)INTEGER; $ maps each graph edges
                                             $ into their lengths

        estimate: min_distance: map(node)INTEGER;
                $ Maps each node into its estimated distance from -start-

        min_estimate: INTEGER; $ shortest estimated distance from
                               $ -start- to any node not yet processed
        reached   : SET(node); $ set of all nodes reached so far
                               $ along a path from -start-
        start, next, outer, n : node;

END;

read(length, successors, start); $ initially, only -start- has been
                                 $ reached
all_nodes := DOMAIN successors + RANGE successors;

reached := {start};

$ Estimate the distance to the nodes that are adjacent to -start-.

(FOR next IN successors{start})
    estimate(next) := length(start,next);
END FOR;

min_distance := {};

(WHILE reached /= all_nodes)

    $ Among the nodes that have not been reached yet, find the one
```

```
    $ whose estimated distance to -start- is the smallest.

    min_estimate := MIN/[estimate(n) : n IN all_nodes | n NOTIN reached];
    ASSERT EXISTS next IN all_nodes | estimate(next) = min_estimate;


    $ The minimum estimate is the shortest distance to next, which is
    $ now considered reached.
    reached WITH:= next;
    min_distance(next) := min_estimate;

    $ Update the estimate for all the nodes adjacent to -next-.  A
    $ path through -next- may yield a shorter distance than that
    $ estimated previously.

    (FOR outer in successors{next} )

        IF estimate(outer) = OM THEN
           estimate(outer) := min_estimate + length(next, outer);
        ELSE
            estimate(outer) MIN:= min_estimate + length(next, outer);
        END IF;

    END;

    END FOR;

END WHILE;

(FOR n IN all_nodes)
  print('The shortest distance from start to ', n , ' is ',
                min_distance(n));
END FOR;;

END PROGRAM;
```

10.4 Basing Declarations

In section 10.1 we remarked that the execution of SETL programs is slowed by two kinds of inefficiencies :

a) Inefficiencies associated with type-checking: every SETL operation is preceded by a test to determine the type of its arguments.

b) Inefficiencies associated with the use of sets and maps: every membership test, every set insertion, every map retrieval or modification requires the calculation of a hash code, followed by a retrieval from a hash table. In what follows we will refer to this sequence of actions as a hashed search.

Inefficiencies of type a) can be corrected by suplementing a SETL program with type declarations, as described in Sec.10.3 . We therefore turn our attention to the means available to corect inefficiencies of type

b).

We begin with the following obvious remark: many programs that use sets and maps search repeatedly for objects that they need to access. As an example, consider the following typical fragment:

(1)
```
S := {....};      $ Some set former expression.
M := {};          $ An empty map.
(FOR x In S)
 M(x) := g(x);        $ Compute map M, whose domain is S;
                      $ g is some defined function.
END;
```

Note that this code performs two hash searches for every element x of the set S: one when S is built, and the second when M is built (i.e. when an element x of S becomes an element of the domain of M). This situation is fairly typical, and it illustrates the kind of redundancy that we want to minimize.

The following somewhat more subtle example shows another aspect of the problem of redundant hashed searching. Consider a set intersection operation:

```
S3 := S1 * S2;
```

The way in which the SETL run-time system evaluates this is best described by the following code fragment:

(2)
```
S3 := {};
(FOR x IN S1 | x IN S2)
    S3 WITH:= x;
END;
```

This means that an element which is in the intersection of S1 and S2 will be searched for twice: first when it is tested for membership in S2, then again when it is inserted into S3. Moreover, a hashed search will also have been performed when S1 was built. Thus, as a single value is inserted into and retrieved from various composite objects, it becomes the object of repeated, redundant searches.

It should be clear at this point that these repeated searches can be eliminated if we somehow 'save' the location of objects so that they can be accessed repeatedly without the need to search for them every time. It is also characteristic of the examples presented above that some of the objects which play a role in them appear in several hash tables and must be searched for in all of them. This last remark suggests that such objects should be kept in one location, and that every use of the object should make use of a pointer to this location, so that no redundant searching will be required.

In other words, if we remember where we leave things, we won't waste time looking for them every time we need them !

To achieve this effect, the data representation sublanguage of SETL uses a special kind of set, called a base set, or base for short, in which such shared values can be stored.

10.4.1 Base Sets

Base sets are special data structures which contain values that are likely to be referenced repeatedly and to be parts of several composite objects (sets, maps, and tuples). Base sets are sets, but sets of a very special nature, which cannot be used in the same way as other sets in SETL. Since they are sets we will speak of the 'elements' of a base, but since they are special we will not apply any of the standard set operations to bases: bases are only introduced to minimize the number of hash searches that must be carried out during program execution and improve the representation of other composite (set, map, and tuple) values.

Bases are introduced into a SETL program by means of declarations of the form:

(1) BASE B;

or

(2) BASE B : <mode>;

Examples of the more specific declaration form (1B) are

```
BASE all_strings:STRING;

BASE all_nodes,all_records:ATOM;
```

The form (1) declares that B is a base whose elements have unspecified type. Form (2) specifies that B is a base whose elements are of type <mode>. We can also introduce several bases at once by writing

(3) BASE <name list> : <mode>;

The <mode>s that can appear in (2) and (3) include those described in Sec. 10.3. Additional modes, to be discussed below, arise from the existence of bases themselves. In particular if x is a variable whose value is expected to appear as part of several composite objects, then we can declare x as follows:

(4) x : ELMT B;

This declaration states that every value assumed by the variable x in the course of program execution will be represented by a pointer to an element of the base B.

Bases declared in SETL programs are used only to define the modes of based objects. They are never explicitly manipulated by the program, and cannot appear in expressions or executable statements. We emphasise again that they serve only to state the existence of significant relationships among actual program objects. These relationships are defined by means of based declarations, and thus, directly or indirectly, in terms of modes of the form (ELMT B).

The effect of a declaration of the form (4) is the following: whenever the variable x is assigned a new value, this value is automatically inserted as a new element of the base B. The new value is placed in a special structure, called an element block of the base B, which contains several pieces of information that pertain to the current value of x. Subsequent references to this value can then use pointers to the element block thereby created.

The information contained in a element block is the following :

a) The value of the element.

b) A system-assigned numerical index, which is uniquely associated with this element. In effect, these indices 'number' the base elements. We will see later that the existence of this numbering allows us to use particularly efficient representations for certain other based objects.

c) Several supplementary storage locations, can also be allocated in each base block. These can be used to hold information about other sets and maps in which the value represented by the element block appears.

To explain the efficiency gains attainable by the use of based repesentations, we will first explain the basing declarations that are available for sets and maps. We discuss based maps before based sets, because the efficiency gains obtained for maps are particularly easy to describe.

10.4.2 Based Maps

If the domains of several of the maps appearing in a program are expected to overlap (i.e. if these maps are likely to be defined on some of the same values) then it may be appropriate to declare a common domain base for these maps. Similarly, if a set is expected to overlap with the domain of a map, it is often advantageous to specify a common base for the set and the map. This is done for maps as follows. Let B be a base introduced by one of the declarations (1)-(3) listed above. Then the declarations:

```
f : SMAP(ELMT B) <mode1>;
g : SMAP(ELMT B) <mode2>;
h : SMAP(ELMT B) <mode3>;
```

state that f,g and h are single-valued maps, whose domain elements are elements of the base B, and whose range elements have other secified modes.

The element block structure described in the previous paragraph allows the maps f, g and h to be represented efficiently, in several ways:

a) In the element block of B corresponding to a given value v, we can allocate storage to hold the values of f(v), g(v) and h(v). If this is done, the structure of each element block of B will be as follows:

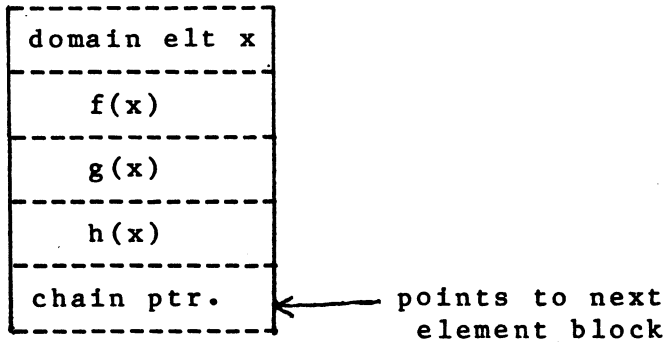


Fig. 10.5 A simple 'element block' in the Based Representation of Three Maps.

Suppose that f is represented in this way, and that x has been declared to have ELMT B representation, so that it will be represented by a pointer to an element block. If, during program execution, we need to evaluate f(x) for a value x which is already an element of B, then we can simply retrieve the value of f(x) from the element block for x. This evaluation of f(x) amounts to just one machine-level pointer reference operation, and is thus considerably faster than a hashed search. Hence representation in the manner shown above is the most efficient one to use for maps which are manipulated exclusively by simple storage and retrieval operations.

Because in this representation map values are stored in immediate proximity to the domain value to which they correspond, this map representation is called LOCAL representation. To ensure that a map is represented as a local map, it must be declared as follows:

(5) f : LOCAL SMAP(ELMT B) <mode>;

The following figure show additional details of data structure introduced by the BASE declaration (1) and by additioal declarations of the form (5).

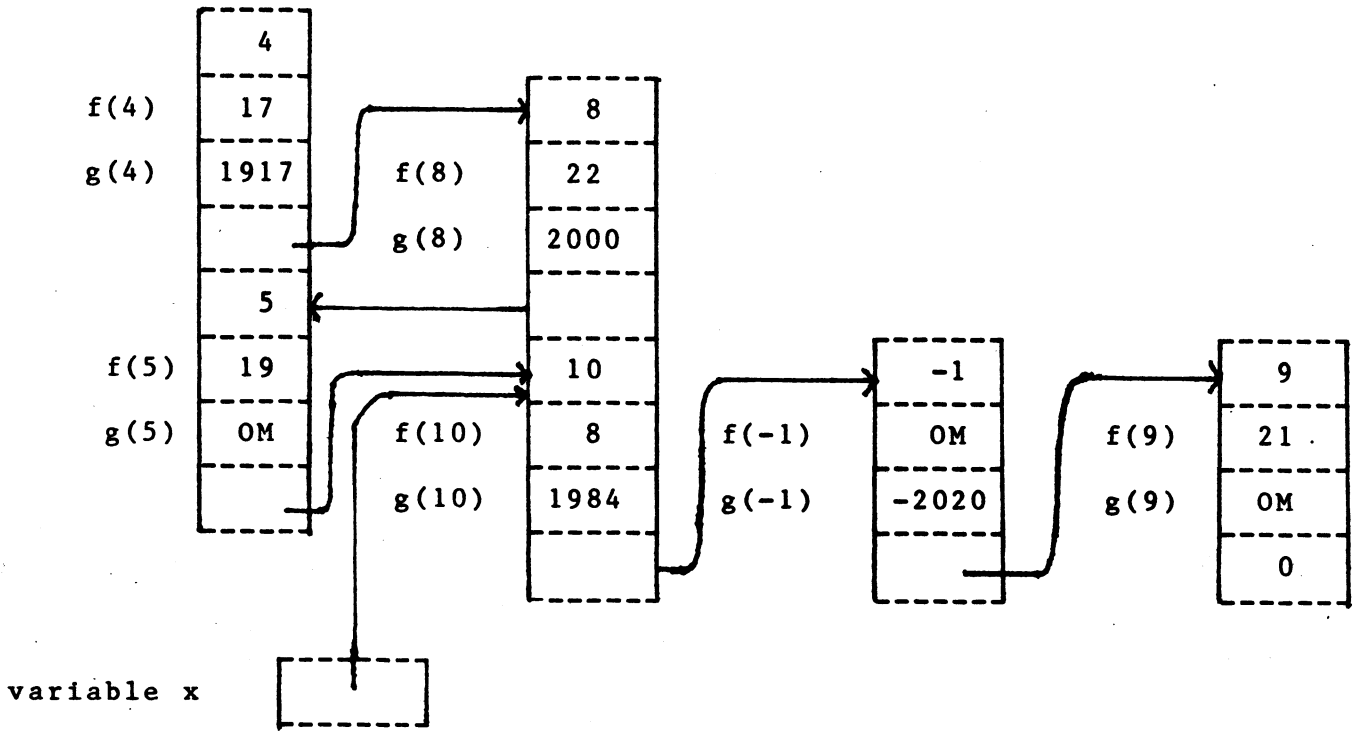


Fig. 10.6 'Base' table which stores the representation of two maps f and g. The variable x is represented here by a pointer to te appropriate block in this 'base'.

LOCAL map representations handle storage and retrieval operations efficiently, but are inefficient for some other purposes. For example, the fact that the range values of a local map are spread over the element blocks of the base maps makes it time-consuming to incorporate a local map as a part of some other composite object (say a tuple of maps). Building the range of F is also time-consuming if F is represented locally. Moreover, iterations of the form

(FOR y = F(x))...

will also be inefficient if F is defined for only a few of the elements of its declared base B . This is because such an iteration must examine each element of B to see whether F is defined for it. Thus LOCAL basing is generally not the ideal way of dealing with maps which need to be made parts of larger composites, iterated over, etc. To handle such situations effectively, other based representations are available.

b) We therefore pass to discussion of a second form of based representation, whose use is advantageous in some of the situations discussed above, in which LOCAL based representation leads to inefficiencies. This second form of based representation is called the REMOTE based representation. It exploits the fact that each element block in a base contains a numerical index that identifies the value that the block represents. The availability of this numerical index makes it possible to store the range values of a REMOTEly based map f in a tuple t. Suppose, to be specific, that ix is the index stored in the block that holds the value x. Then the value of f(x) is

held in the ix-th element of the tuple t. In this case, f(x) is is retrieved as follows:

i) Using the pointer in x, retrieve ix from the element block for x.

ii) Add ix to the starting address of the tuple t that holds the range of f, retrieve the ix-th component of this tuple, and return its value.

This sequence of operations is considerably faster than a hashed search, even though it is slower than access to a LOCAL map. (We call this type of map representation REMOTE because it stores range elements at some distance from the corresponding domain elements). To specify that a based map is to have remote representation, we simply declare it as follows :

(6) g : REMOTE smap(ELMT B) <mode>;

c) SETL provides a third based representation of maps, called to SPARSE representation, which is motivated by other considerations of storage and iteration efficiency. The two representations described so far, LOCAL and REMOTE, are both characterized by the fact that to hold the values of f(x), a storage location must be allocated for each element of the base, regardless of whether f(x) is defined or is OM. In the local case, this location is allocated directly in the element block of x; in the REMOTE case, this location is the array component location corresponding to the identifying index of x. In both cases, if f is sparsely defined over its base, then a substantial number of storage locations will be wasted. (By 'f is sparsely defined' we mean that f(x) /= OM only for a small percentage of all the values x in the base of f). For such sparse maps, the third, so-called SPARSE, based representation may be advantaeous. To give a map f this representation, we declare it as

(7) f: SPARSE MAP(ELMT B) <mode>

The SPARSE map representation uses a hash table, very much like that used to standard unbased maps. However, the SPARSE map representation does not hold the value of each of its domain elements, but rather represents each domain element x of F by a pointer to the element block in B that represents x. The distinction should be clarified by the following figure, which compares the organization of unbased and sparse maps :

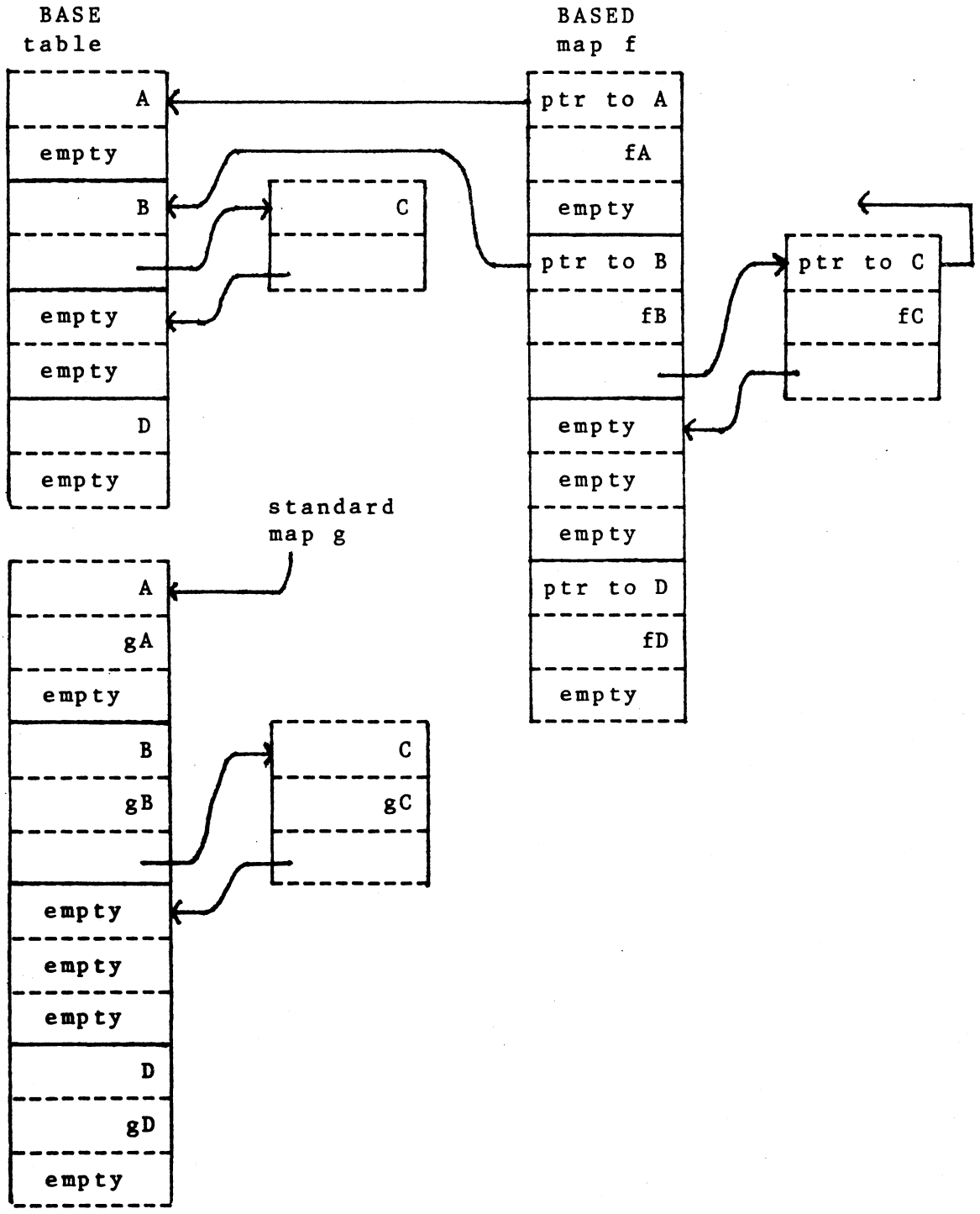


Fig. 10.7 Internal representation of unbased and of Sparse maps.

Evaluation of f(x) for a sparse map is distinctly less efficient than for a remote map, but somewhat faster than for an unbased map. As already noted , an important reason for using sparse maps is storage efficiency. Map iteration is an operation that also benefits from the use of the sparse representations. For a local or remote map, the iteration

```
(FOR y = F(x))
```

requires a full iteration over the base F, which then bypasses the elements of the base for which F is undefined. In other words, the iteration is performed as if it was written :

```
(FOR x in B | (y := F(x)) /= OM)
```

If F(x) = OM for most elements of B it is plain that this iteration will examine a large number of useless elements. If F is represented as a sparse map, its domain is directly available, and no useless elements need to be examined.

The qualifiers LOCAL, REMOTE, and SPARSE can be omitted from a basing declaration. The 'default' if all are omitted is SPARSE, that is

```
f:  SMAP(ELMT B) <mode>
```

and

```
f:  SPARSE SMAP(ELMT B) <mode>
```

are equivalent.

10.4.3 Based Representations for Sets

Three types of based representations are available for sets; these representations parallel the ones for maps which we have just described. Based sets can therefore be described as having LOCAL, REMOTE, or SPARSE representations.

a) Suppose that the following basing declaration is given:

```
S1, S2, S3 : LOCAL set(ELMT B);
```

Then S1, S2 and S3 are stored internally as follows: in the element block of each element x of B, we reserve one bit to indicate the membership of x in S1, another bit to indicate membership in S2, and so on. These bits are allocated in fixed locations within every element block of B.

When this representation is used, then the test (x IN S) and the set operations (x WITH S) and (S LESS x) are handled in a particularly efficient way when x is an element of the base B : in this case, x is represented by a pointer to its element block, and all that is needed is examination or modification of a single bit at a fixed position in that block, which can be accomplished in very few machine operations. The set representation just described is also storage-efficient, because it uses only 1 bit per element of a based set, in contrast to the several words per element which are required in an (unbased) hash table.

For sets that are constructed and accessed by the above operations exclusively, the LOCAL representation just described is to be prefered over others. However, the drawbacks of this representation are similar to those mentioned above for local maps. It is well to discuss the point in more detail. Certain crucial facts affecting the efficiency of based representations derive from particular semantic rules of SETL. As already

emphasised, the use of based representations is not allowed to change the meaning of a SETL program: basing declarations can only affect its efficiency. The elaborate machinery of pointers, indices and bit positions that we have been describing can in no way affects the semantics of the original (undeclared) program to which such declarations may be added. This means in particular that the use of basings must cause no non-standard side effects. Recall that the semantic definition of SETL requires that the fragment :

```
S1 := {1};
S2 := S1;
S1 WITH:= 2;
```

gives S2 the value {1} , and that the insertion of 2 into S1 which follows subsequently does not affect the value of S2. The original value is S2 is preserved because, logically speaking, it is given a 'personal' copy of the value {1}, rather than 'sharing' this value with S1. (In fact this copy is created right before S1 is modified, but this is an implementation detail). Now if S1 is a LOCAL based set and S2 is not, then producing a copy of S1 is a potentially expensive process which requires full iteration over the base B to extract the elements of B which are in S1. Furthermore, if S1 is itself inserted into some composite object, as in

```
SC WITH:= S1;
```

it must copied first, in order to prevent accidental sharing of values (and potential modification) between S1 and the (now anonymous) element of SC which holds the value of S1. Because of this requirement, LOCAL sets can become sources of run-time inefficiencies whenever they must themselves be shared . Hence, LOCAL sets should only be used for sets that only appear in elementary insertion, deletion and membership tests, and that do not become themselves elements or components of larger composite objects.

b) The declaration :

```
R1, R2, R3 : REMOTE set(ELMT B);
```

gives R1, R2, and R2 a representation which is particularly efficient for global set operations, i.e. union, intersection, set difference and set assignment . This representation, which is analogous to the REMOTE representation for maps, (and which is called the REMOTE set representation) makes use of the indentifying index present in each element block. More specifically, each of the sets R1, R2, R3 appearing in the preceding declaration is represented by what is conceptually an array of zeroes and ones, but which at the implementation level is actually a sequence of machine bits, occupying one or more words of memory. These bits are in one-to-one correspondence with the elements of the base B : the element block whose index is i corresponds to the i-th bit in this bit-vector. If the value in element block i is an element of the set R1, then the corresponding bit in the bit-vector representation of R1 is on, but otherwise off. The ith bit position in the representation of R2 and R3 is used in the same way to indicate membership of an element of B in each of these based sets. If R1, R2 etc. are given REMOTE representation, then the elementary set operations (x IN R1, R1 WITH x, R1 LESS x) can be performed in the following manner, assuming as before x is an element of the base B :

a) Retrieve the index i of x from the element block of x in B.
b) Use this index to access the i-th bit in the bit-vector which represents R1.
c) Return the value of this bit (or modify this bit if a WITH or LESS operation is being performed).

This process is somewhat more time-consuming than the same operation on local sets, but it is considerably faster than the same operation on an unbased set.

The efficiency gains obtained for certain global set operations (union, intersection, etc.) are particularly substantial when the REMOTE set representation is used. Suppose, for example, that R1, R2, R3 have the representation shown above and that we want to evaluate the union

```
R3 := R1 + R2;
```

Then the REMOTE representation of R3 can be calculated as follows: the i-th bit in the representation of R3 (corresponding to some element x of the base) should be on if x is either in R1 or R2, i.e. if the i-th bit of R1 or the i-th bit of R2 is on. The machine-level boolean operation OR performs exactly this bit-by-bit operation on a full machine word of bits in a single step. Thus, on a 32-bit machine, the OR-ing of two bit-vectors of size 1000 will take less than 50 machine operations. By contrast, the union of two unbased sets of size 1000 will require 1000 membership tests and up to 1000 hash table insertions. Similarly the intersection operation on remote sets reduces to the machine-level AND operation, with the same gains in speed. Thus, for large sets on which union and intersection operations are frequently performed, REMOTE representations are extremely efficient; and the efficiency gains attained by this representation are larger, the larger the sets that enter into these operations.

c) Finally, for representing sets that are relatively sparse (i.e. have a cardinality which is much smaller than that of their base set) and over which iterations are frequently performed, a SPARSE set representation is provided. The declaration

```
SP1, SP2, SP3 : SPARSE SET(ELMT B);
```

specifies that SP1,SP2 and SP3 are to be represented by means of hash tables, in which, rather than storing the values of the set elements we keep pointers to these values, i.e. pointers to the element blocks in the base B that holds the actual element values.

As in the map case, the qualifiers LOCAL, REMOTE, and SPARSE can be omitted, and SPARSE is the default: If no qualifier appears in a basing declaration for a set, it is equivalent to specifying a SPARSE representation for it.

10.4.4 Basing Declarations for Multivalued Maps

e saw in Sec. 10.4.2 that declaring a based representation for a single valued map relates the domain of the map to some base in which the domain elements of the map are automatically inserted. A similar

representation is available for multivalued maps, i.e. multivalued maps (which is to say MMAPs) can be given LOCAL, REMOTE or SPARSE representations. Moreover, it is possible to declare a based representation for the range of a multi-valued map F. The value of F{x} is by definition a set, and therefore the based representation for a multivalued map will generally specify an additional basing which determines the representation of the range sets of F. For example, we can declare

```
successors : LOCAL MMAP(ELMT B) REMOTE SET(ELMT B);
```

this declaration specifies that for each x IN B the image set successors{x} is stored in the element block of x, and that this image set is always to be rpresented as a bit-vector. Similarly, the declaration :

```
successors : LOCAL MMAP(ELMT B) SPARSE SET(ELMT B);
```

specifies that the image set of successors{x} is to be stored as a sparse set, i.e. as a hash table containing pointers to elements of B. Note that the attribute LOCAL cannot be used for image sets of multivalued maps. This follows from our remarks in Sec.10.4.3 on the impossibility of making local objects into subparts of composite structures.

10.5 Base Sets Consisting of Atoms Only

The based data structure shown in Figs. 5 serves to support two fundamental operations:

(a) The ability to locate an item x in a base by searching a short list of items, from a staring list position which can be calculated easily if the value of x is known;

(b) The ability to iterate over all the elements in the base.

Operation (a) is only required when an object x is converted to ELMT B representation and we need to determine if x has already been inserted into the base B. Hence, if the only elements ever inserted into B are atoms, and if all of these are created by easily located calls to the NEWAT operator, then the seaching operation (a) is not required, since each call to NEWAT produces a unique object. Hence the blocks constituting such a base can be stored as a simple list. The elemets of this list only need to be linked together if iteration over some set having SET(ELMT B), SMAP(ELMT B), or MMAP(ELMT B) is necessary. If this is not the case, then no links are necessary; the element blocks of B are then independent.

To allow declaration of these important special cases, the data representation sublanguage allows the keyword PLEX to be prefixed to base, as in

(1) B: PLEX BASE;

If B is a PLEX BASE, then only atoms cn be given ELMT B representation.

10.6 Constant Bases

A constant set, introduced by a constant declaration (see Section 6.2), as in

```
CONST colors={red,blue,green};
```

can be used as a base if it is declared as such by writing

```
BASE colors;
```

Elements of such a base B, i.e. values x having the representation ELMT B, can be represented in fixed small number n of bits. Specifically, n must be at least as large as the logarithm of the number #B of elements in B. Internally, a constant base B is represented by a contiguous series of blocks, and an element x having the representation ELMT B is represented by a short integer index that locates the block corresponding to x. Remote subsets s of B can then be represented by bit-vectors, often no more than one machine memory word in length. In this case, the membership test x IN s will be particularly fast if x and s have the representations ELMT B and SET(ELMT B) respectively, since then the representation of x is simply the index of the bit in the vector representing s which determines the boolean result of the test x IN s. A similar remark applies to maps f having the representation SMAP(ELMT B) or MMAP(ELMT B). Moreover, since the internal representation of any value of mode ELMT B can be quite short, it is possible to pack several ELMT B values into a single machine word. To achieve this , one uses the representation qualifier PACKED, in the manner explained below.

10.7 The Representation Qualifier PACKED

The keyword PACKED can be prefixed to SMAP or TUPLE modes. That is, we can write

```
(4)     f:PACKED LOCAL SMAP(ELMT B)mode';
        g:PACKED REMOTE SMAP(ELMT B)mode';
        h:PACKED TUPLE(mode');
```

etc. However, for these constructs to be legal, the mode indicator -mode'- shown in (4) must designate some packable mode, i.e., some mode of values which can be represented in less than a full machine memory word. (Note that the machine words of typical present-day computers generally contain between 32 and 64 bits of information. Thus, for example, if a quantity can be represented in just four bits, i.e., if it can take on at most sixteen different values, then between eight and sixteen quantities of this kind can generally be represented by parts of a single machine word).

Modes of the two following kinds are packable in this sense:

(i) The mode ELMT B, where B is a constant base (see Section 10.6).

(ii) The mode INTEGER n1..n2 (see Section X), provided that the interval [n1..n2] over which integers of this mode range is sufficiently small.

If the mode' appearing in a declaration (4) is packable, then the SETL compiler will know how many bits are required to represent values having this mode. It will then be able to store several packable local map values like f(x) (cf. (4)) in a single machine word of the base block of B representing an ELMT B value x. Moreover, in the vector (cf. Fig.) used to store range values of a PACKED REMOTE map (like the g of (4)), it will be possible to store several map values per machine word. Similarly, several tuple components of a PACKED TUPLE (like the h of (4)) can be stored per machine word.

This use of packed storage saves memory space, thereby reducing the space needed to run your SETL program. On the other hand, the number of machine cycles needed to run the program will rise slightly, owing to the necessity to convert quantities between their packed and unpacked forms. However, since the cost of such conversion is small (provided that effective representations are chosen for all the variables appearing in a program; see Section 10.8), the storage economy obtainable by packing data where possible can far outweigh the modest execution-time costs which packing incurs.

10.8 Guidelines for the Effective Use of the Data Representation Sublanguage

By adding appropriate data representation declarations to your program, it will often be possible to increase its efficiency substantially. Moreover, a SETL program for which a well thought-out set of representations has been specified will often constitute a detailed blueprint from which an efficient program in some lower-level language such as PASCAL, PL/I, or Ada can be generated, manually but in a mechanical spirit. In this section we will explain the principles governing effective choice of data representation declarations, note some of the restrictions governing the use of the representation sublanguage, and also point out some of the efficiency pitfalls of which you should beware.

As already noted, the main aim of the data representation sublanguage is to speed up functional evaluations f(x) and f{x}, also membership tests x IN s, by ensuring that for as many such evaluations as possible x has ELMT B representation and f has SMAP(ELMT B) representation (or MMAP(ELMT B)) representation (or s has SET(B) representation), where x and f (or x and f) are based on the same base B. On the other hand, to attain a net gain using this approach, we must be sure that the cost of converting elements x, maps f, and sets s to their based representations does not outweigh the advantage gained by use of such representations. We must also be sure that our choice of representations does not cause excessive object copying to take place. (The circumstances under which object representations are copied during program execution will be described in more detail below).

Objects are converted between different internal representations in the following circumstances:

(i) When a SETL value is read from an external file by a READ statement and assigned as the value of an identifier x for which some based representation has been declared, the new value of x will be converted, from the standard representation in which it is first read, to the representation declared for x. A reverse conversion takes place whenever a PRINT statement is used to move a value x having some specially declared representation to an external

file. There is little you can do about conversions of this kind, whose cost is in any case bounded by the amount of input and output which your program performs.

(ii) Whenever a value x having one representation is passed by an assignment y:=x to another variable y for which a representation has been declared, x is converted to the form declared for y. A similar conversion takes place whenever x is made part of a composite object y (i.e. a set, tuple, or map), by an assignment y:=y WITH x, y(z):=x, y(x):=z, etc. In these cases, x is converted to the form expected for the part of y which it becomes. For example, in the case of y:=y WITH x, if y has been declared to have the representation SET(mode'), then x will be converted to the representation mode'. In the case of y(z):=x (resp. y(x):=z), if y has been declared to have the representation MMAP(mode)mode' or SMAP(mode)mode' (resp. MMAP(mode')mode or SMAP(mode')mode), then x will be converted to the mode' form. (A fuller list of conversion rules of this kind is given at the end of this Section).

(iii) Values extracted from composite objects y will initially have representations deduced from the representation declared for y. For example, if s is declared to have the mode SET(ELMT B), then the iterator FORALL x IN s... will produce elements of s, each such element initially having ELMT B format, and assign them successively to x, converting them to the form declared for x if necessary. Similarly, if f is declared to have SMAP(ELMT B1)ELMT B2 representation, then evaluation of f(x) will require that x be converted to ELMT B format, and f(x) will yield a value of mode ELMT B2. If x had some other format immediately prior to the valuation of f(x), or if we use an assignment z:=f(x) involving a variable z that has been declared to have some representation other than ELMT B, then appropriate conversions will be forced.

(iv) The conversions performed when we execute assignments (i) are also performed in connection with expressions, such as existential and universal quantifiers, having assignment-like side effects, and also in connection with iterators. For example, if s is declared to have SET(ELMT B) representation, but x is declared to have some representation other than ELMT B, then evaluation of an existential quantifier like

```
...EXISTS x IN s | C(x)...
```

will repeatedly extract elements from s (in ELMT B format) and convert them to the representation declared for x.

(v) Whenever procedures and functions are invoked, their actual arguments are converted to the representations declared for the corresponding formal parameters. Moreover, if a function returns a value having one representation but this value is assigned to a variable for which some other representation has been declared, a conversion will take place.

To minimise these conversions, you need to choose representations for the various data items appearing in your program which make conversion unnecessary. To accomplish this you will need to survey the undeclared form of your program carefully, noting the manner in which each variable is used. The appearance of an assignment x:=y will suggest that x and y should be given the same representation; tests x IN s will suggest that s should have

the representation SET(mode), where x has the representation -mode-; map evaluation y:=f(x) will suggest that f should have the representations SMAP(mode)mode' where x has -mode- and y has -mode'- representation; etc. Chains of deductions of this sort, together with a bit of reflection about the abstract nature of the various objects which your program is manipulating, will generally lead without undue difficulty to a 'harmonious' set of representations avoiding unnecessary conversions. Note that both conversions within single PROCEDUREs, and conversions of arguments forced when one PROCEDURE invokes another, are to be avoided. If there remain some conversions which cannot be avoided, care should be taken that these conversions take place at infrequently executed points in your code.

10.9 Exercises

Ex. 1 Develop an effective set of representation declarations for the buckets-and-well program shown in Section 4.3.1.

Ex. 2 Develop an effective set of representation declarations for the Eulerian path procedure shown in Section 11.1.

EX. 3 Develop an effective set of representation declarations for the topological sorting procedure shown in Section 7.2.

10.10 Additional Remarks on the Effect of REPR Declarations

If a set s is declared to have local representation, then each block of the base B shown in Fig. 10.6 is enlarged by an extra machine word, and a specific bit in all these words is reserved to indicate whether or not the element x represented by the block belongs to s. If (s IN s) is TRUE, then in the block representing x this reserved 's-bit' will have the value 1; if FALSE, then this bit will have the value 0. It is then; clear that the test x IN s can be made very rapidly if x and s have the repesentations (4) and (5) respectively. Moreover, the operations s WITH := x and s LESS := x will be quite fast, since both of these set-theoretic operations can be executed by using a machine level operation to change just one bit in the block located by x.

Note that sets s declared to have the representation (5) are represented internally in 'distributed' rather than 'concentrated' fashion. That is, s is represented by a scattered set of 'flag bits', one bit per block of the base B, rather than by a hash table of the more concentrated form shown in Fig. 3. The figurs following just below shows a base B and the representation of two sets s1, s2 declared to have LOCAL SET representation:

(7) s1, s2: LOCAL SET(ELMT B)

The figure assumes that s1={3,17,201} and that s2={201,48,722}, and that the rightmost bit in the second word of each entry in the base table is being used to indicate membership in s1, while the bit next to it is used to indicate membership in s2.

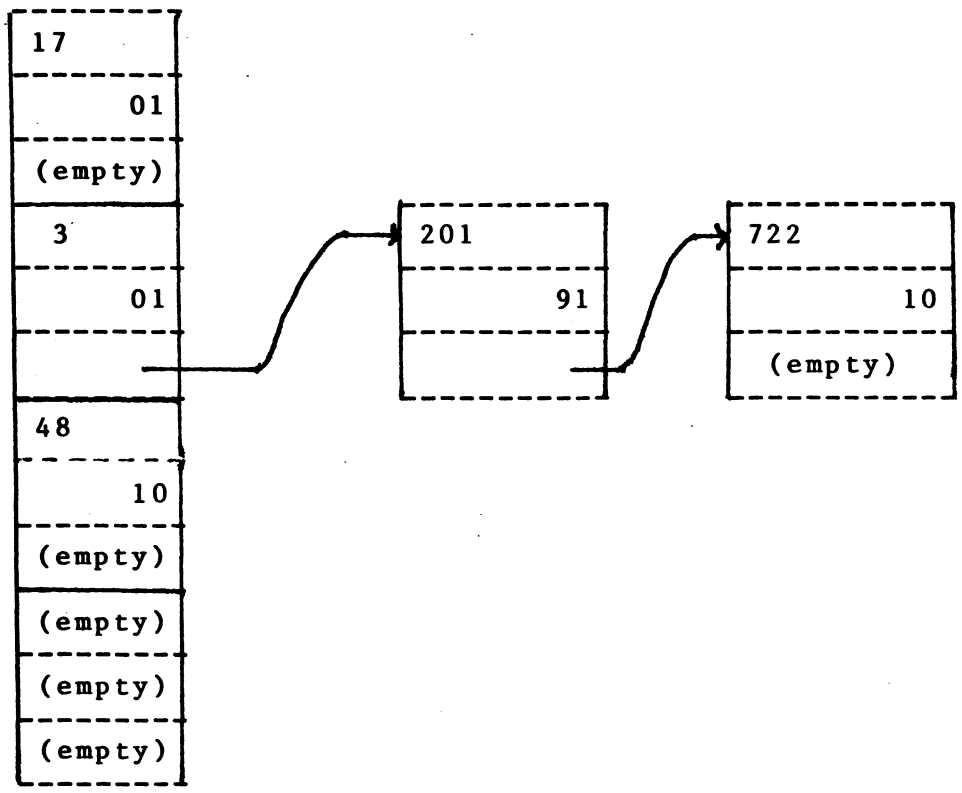


Fig. 10.8 Internal representation of two sets s1, s2, both declared to have LOCAL SET(ELMT B) representation.

Note that, although use of LOCAL SET(ELMT B) representation for a set s will speed up the test (x IN s) if x has ELMT B representation, it may slow down iterations of the form

(6) FOR x IN s...

substantially. This is because the representation of s shown in Fig. 3 makes it possible to iterate rapidly over the elements of s and no other; in contrast, if s has the representation shown in Fig. 3, we must handle iteration over s by iterating over all the blocks of B, but then skipping past those which do not represent elements of s. If only a small percentage of the blocks of B represent elements belonging to s (whih can easily happen, for example, we may declare s,s2:SET(ELMT B), and s2 may have many more elemets than s) then the iteration (6) can be slowed considerably.

In some cases, it is better to represent a set s by flag bits that are grouped together than by distributed bits. To do this, a declaration of the form

(8) s: REMOTE SET(ELMT B)

is used. In the presence of the declaration (8), each entry E of the table representing the base B will contain an integer index i, issued by the SETL run-time system when the element x represented by E is first inserted into

B. The set s is then represented by a sequence of bits, grouped together into one or more machine words. If the element x of B has been issued index i by the SETL run-time system, then the i-th among these bits will be 1 if (x IN s) is TRUE, 0 of (x IN s) is FALSE. Fig. 7 shows this form of set representation.

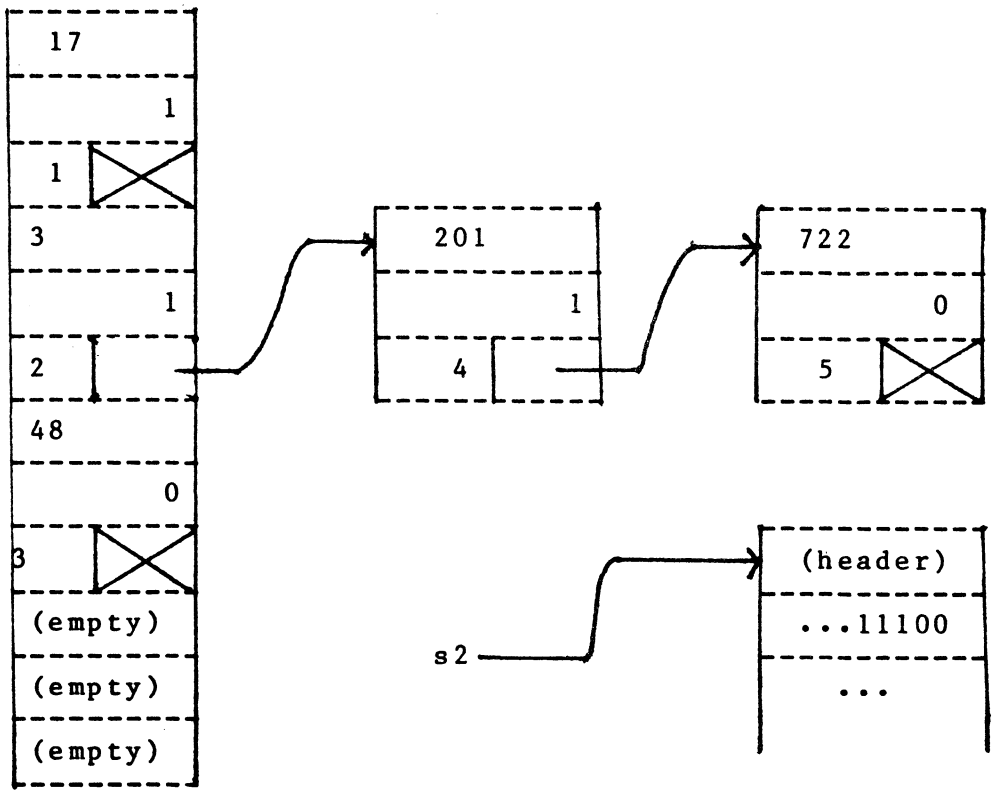


Fig. 10.9 Internal representation of the sets s1 = {17,3,201} and s2 = {201,48,72}, with declared representations s1: LOCAL SET(ELMT B) and s2: REMOTE SET(ELMT B).

Note that the left half of the third word of each block of B is asumed to contain the index which the SETL run-time system assigns to the block.

Even though the test x IN s is slowed down slightly if we give s REMOTE SET rather than LOCAL SET representation, the REMOTE SET representation illustrated in Fig. 7 will sometimes have substantial advantage over the corresponding LOCAL set representation. First of all, if s1 and s2 both have REMOTE SET(ELMT B) representation, then Boolean combinations of s1 and s2, e.g. s1 + s2, s1 * s2, and s1 - s2 can be formed very rapidly using the machine level Boolean operations described in Section 9.3.2; these oper machineations handle a word-full of Boolean bits per operation cycle. An even more crucial advantage is that indefinitely many (exact or approximate) copies of a set s having REMOTE SET representation can be formed simply by allocating additional copies of a vector of bits like that shown in (the lower right-hand corner of) Fig. 7. Such easy copying is not possible for sets having LOCAL SET representation, since the flag-bit positions within

base blocks used to indicate membership in such sets must be allocated in advance. It follows in particular that the components of a set of tuples, or the members of a set of sets, can be given REMOTE SET but not LOCAL SET representation. Note that to speed up a membership test like x IN t(i) or to ensure that an iteraion like

(FOR s IN set_of_sets) IF x IN s THEN...

runs at high speed, we may be need to give the components of -t- or the members of -set_of_sets-, based form. as stated we must then use the REMOTE SET representation.

The REMOTE form of representation is available for maps f as well as sets s. To give a map f this representation, one writes

(9a) f: REMOTE MMAP(ELMT B)
(if the map f might be multi-valued), or
(9b) f: REMOTE SMAP(ELMT B)

(if f is known to be single-valued). The range values of a map represented in this way are gathered together into a continguous array of memory cells, the i-th memory cell holding the value (or, in the MMAP case, set of values) associated with the domain element whose index is i. This map representation is shown in Fig. 8, which should be compared with Fig. 5. The map represented in Fig. 8 is f={[17,71], [3,33], [201,102]}, and is assumed to have REMOTE SMAP(ELMT B) representation.

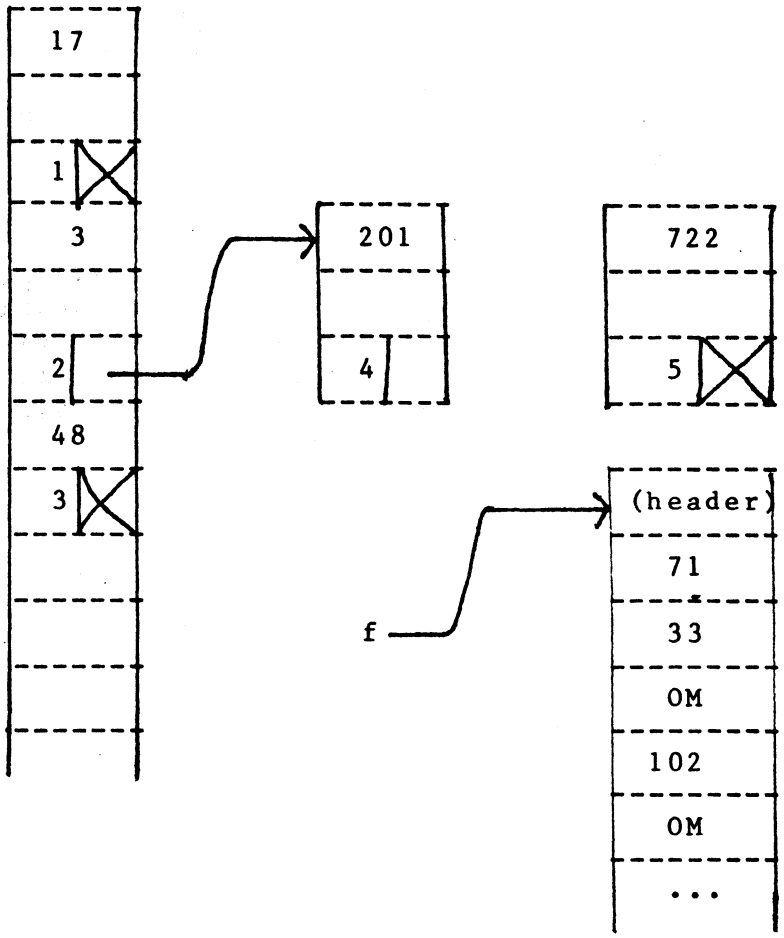


Fig. 10.10 Internal representation of the map f={[17,71],[3,33],[201,102]} in REMOTE SMAP(ELMT B) form.

The left half of the third word of each block of B is assumed to contain the index which the SETL system assigns to the block. Note that 48 and 722 are assumed to be base elements but not elements of DOMAIN f.

The advantages as disadvantages of REMOTE and distinct from LOCAL map representation are similar to those of the corresponding set representations. In particular, multiple copies of a map f having REMOTE MMAP or REMOTE SMAP representation can be formed simply by allocating a vector of range values like that shown in Fig. 8. This is not possible for sets having LOCAL map representation.

In order to make use of the based representations in Figs. 5-8, the SETL run-time system code must ensure that every element x belonging to a set s with SET(ELMT B) or to the domain of a map f with MMAP(ELMT B) or SMAP(ELMT B) representation is assigned an entry in the base table B. This is accomplished by keeping B under systen rather than programmer control. Then, whenever an operation like s WITH := x, s +:= s1, f(x) := y, f{x} := -1, etc. adds one or more elements to s or to DOMAIN f, it is automatically added to B. For this reason, SETL does not allow bases B to be used in the same way as ordinary variables. In particular, base names can appear in declarations like (1-9), but not in ordinary SETL statements

Since both the LOCAL SET(ELMT B) and the REMOTE SET(ELMT B) representation will slow iterations of the form (FOR x IN S)... very substantially if few of the blocks in B represent elements of the set s', the data representation sublanguage provides a third kind of declarable representation for sets, namely

(10) SPARSE SET(ELMT B)

If a set s is declared to have this representation, it will be represented by a hash table having much the same form as the standard representation shown in Fig 3; however, the entries in this table will contain pointers to blocks of the base B rather than standard-form SETL values. This is illustrated by the following figure, which shows how the set s={17,3,201} would be represented if it were declared to have representation (10).

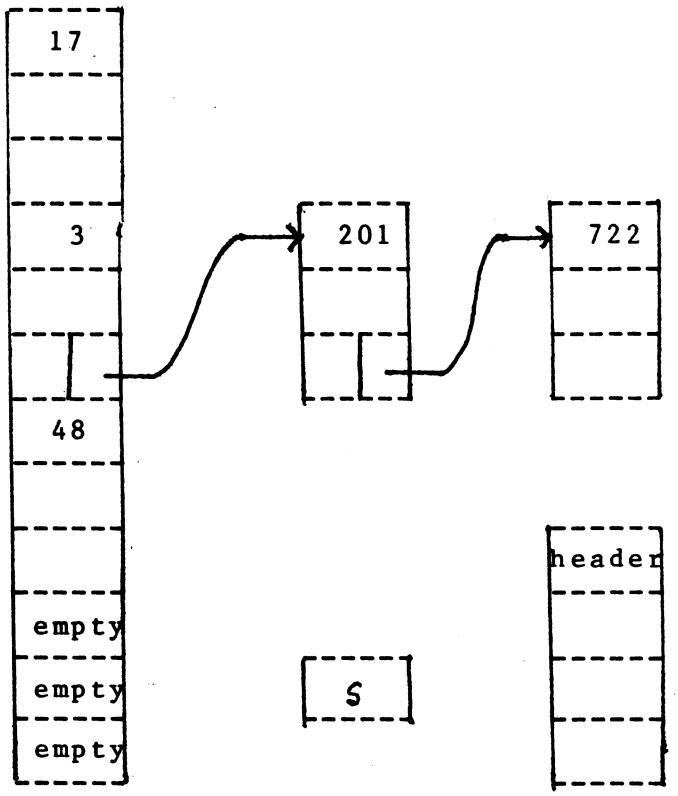


Fig. 10.11 Internal form of the set s={17,3,201}, assuming that the representation s: LOCAL SET(ELMT B) is being used.

Iteration over set s having LOCAL SET(ELMT B) representation will always be efficient, since no elements not belonging to s are examined during such an iteration. Moreover, this iteration will produce items x having ELMT B representation. This makes the SPARSE representation particularly effective for iterating over a set of elements that are to be used as map indices. For example, an iteration of the form

(11) (FOR x IN s) y := f(x);...

will be particularly efficient if the following representtions are declared

for x,s, and f:

```
(12)    x: ELMT b;
        s: SPARSE SET(ELMT B);
        f: LOCAL SMAP(ELMT B);
```

he reader should confirm his understanding of the preceding pages by reviewing the data structures that will be used for x,s, and f in the presence of these declarations and by working out the sequence of machine-level operations that will be needed to handle the code fragment (11) in the presence of the declarations (12).

Map representations akin to the SPARSE SET representation (10) are also available. These are declared by writing

```
(13a)   f: SPARSE MMAP(ELMT B)
or
(13b)   f: SPARSE SMAP(ELMT B).
```

Maps declared in this way are given internal representations much like those shown in Fig. 4, but the domain elements of such maps are represented by pointers to blocks in the base B rather than by SETL values in their standard form. As in the case of SPARSE SETs, iterations over maps having this representations are handled efficiently. Moreover, if f has either of the representations (13a) or (13b), then an iteration like

```
(14)    (FOR y = f(x))...
```

will produce items x having ELMT B representation.

This completes our introductory account of the main facilities of SETL's data representation sublanguage and of the principal advantages and disadvantages of the major set and map representations describable in this sublanguage. Various other features of the data representation sublanguage will be presented in Sections X and Y. The representation language can be used to improve the efficiency of SETL codes, but to achieve this you must devise a consistent pattern of declarations, assigning an appropriate representation to each of the data items used in the code. What one wants are declarations which give ELMT B representation to variables x appearing in contexts like (x IN s), s WITH :=x, s LESS := x, f(x), f{x}, f(x) := y, or f{x} := y, while s and f are given SET(ELMT B) and SMAP(ELMT B) or MMAP(ELMT B) representation. However, the various pitfalls pointed out in the foregoing paragraphs, for example the possibility of showing down an iteration (FOR x IN s)...if s is given LOCAL SET(ELMT B) or REMOTE SET(ELMT B), must be borne in mind. It is also important to note that the efficiency gains obtainable by skillful use of SETL'S data representation sublanguage will be lost if inconsistent or incomplete declarations cause values to be converted between different representations in frequently executed code sections. For example, if s1 and s2 are declared to have different representations, e.g.

```
(15)    s1: LOCAL SET(ELMT B);
        s2: REMOTE SET (ELMT B);
```

then any assignment

```
s1 := s2;
```

or for that matter any operation

```
s1 := s2 WITH x;
```

or

```
s := s1 + s2;
```

will cause a copy of s2 to be converted into the form declared for s1, Similarly, if x and f are declared to have the forms

(16)
```
x: ELMT B1;
f: LOCAL SMAP(ELMT B2);
```

when the bases B1 and B2 are different, then either the operation y := f(x) or f(x) := y will cause a copy of x to be converted into ELMT B form. This conversion can easily get out of hand, and then can cause a program containing representation declarations to be less rather than more efficient than its original declaration-free version.

The SETL measurement facility described in Section Y can be used to determine the actual effect of a given set of representation declarations, and in particular to pinpoint statements at which unexpected object copying or conversion between representations is taking place.

Additional hints concerning effective use of SETL's data representation sublanguage are found in Section 10.8 above.

To conclude this section, we note that a SETL program, supplemented by a carefully worked out set of representation declarations, can be regarded as a detailed blueprint for a lower-level implementation of the same program. Used in this way, SETL serves well as a tool allowing program a skilled designer or design team can convey all the details of his (or their) program design to a larger, perhaps less experienced reprogramming team . This reprogramming team can use some other more efficient language to produce a high-efficiency code from a design written in SETL. The fact that the original SETL code actually executes means that the consistency and completeness of the initial design can be verified by testing rather than by visual inspection or 'design walkthrough' only.

$

CHAPTER 11

THE LANGUAGE IN ACTION: A GALLERY OF PROGRAMMING EXAMPLES

In this, our last chapter, we illustrate the use of SETL by giving a variety of programs which exhibit its features and can serve as useful models of style. Some of the smaller programs present significant algorithms; the larger examples show how more substantial programming problems and applications can be addressed.

Chapter Table of Contents:

11.1 Eulerian paths in a graph

A graph is simply a collection of nodes, pairs of which are connected by edges. (See Section 5.3). Graphs come in two varieties, directed graphs, each of whose edges has a specified starting node and target node, and undirected graphs, whose edges can be traversed in either direction. The most natural SETL representation of a directed graph G is simply a set of ordered pairs [x,y], each such pair representing in edge with starting node x and target node y. It is convenient to represent an undirected graph G in the same way, but in this case the reversed edge [y,x] belongs to G whenever [x,y] belongs to G.

Given an undirected graph G, the Eulerian path problem, named after the famous mathematician Leonhard Euler(1707-1783, 'who calculated as other men breathe') is to traverse all the edges of G exactly once by a single unbroken path p which starts at some node x of the graph, and ends at some

other node y (which might be the same as x). We can think of such a path, called an Eulerian path, as 'using up' edges as it traverses them. Euler used the following arguments to determine which graphs contain paths p of this kind. If a node z along p is different from the starting and ending nodes x and y of p, then immediately after p has reached z along one edge p will leave it along some other edge, and thus p will always 'use up' an even number of the edges which touch any node z of p not equal to x or y. The same remark applies to the starting node x if x = y, but if x and y are different then p must 'use up' an odd number of the edges touching x and an odd number of the edges touching y. It follows that an Eulerian path p which traverses all the edges of G just once can only exist if G is connected and either has no nodes x touched by an odd number of edges, or has exactly two such nodes x,y; and in this latter case every Eulerian path p must start at one of x,y and end at the other.

Suppose, conversely, that G has either no nodes or exactly two nodes which are touched by an odd number of edges. Then we can construct an Eulerian path p as follows. If every node of G is touched by an even number of edges of G, let x be any node of G, otherwise let x be one of the two nodes x, y of G touched by an odd number of edges. Start the path p at x, and extend p as long as possible by stepping from its endpoint along any edge of G that has not been traversed before. Since we consider an edge to be 'used up' as soon as it is traversed, the construction of p uses up more and more edges of G, and therefore must eventually stop. Hence p must be finite. Suppose that p ends at a node y. Clearly all the edges touching y must have been traversed by p, since otherwise p could be extended by some edge. Thus, if the starting node x of p is touched by an odd number of edges, p must end at some other node y which is also touched by an odd number of edges; whereas if x is touched by an even number of edges, then p must return to x and end there. In either case, removing all edges traversed by p from G will leave behind a graph G' each of whose nodes is touched by an even number of edges. If p does not already traverse all the edges of G, then some node z along p will be touched by some untraversed edge. In this case, one can construct a path q by starting from z with this edge and extending q along untraversed edges as long as possible. Since the remarks concerning p apply to q as well, and since q can be regarded as a path in the graph G', and since all of the nodes preceding G are touched by an even number of edges, the path q must both begin and end at z, i.e. q must be a 'circuit'. Hence we can 'insert' q into p, thereby constructing a path which first follows p to z, then follows q until q finally returns to z, and then follows the remainder of p to its end. Call this extended path by the same name p. Repeating the construction and insertion of circuits like q as often as possible, we must eventually built up a path p which traverses all the edges of the original graph G.

The two following procedures realise the Eulerian path construction described in the preceding paragraphs. Procedure build_path starts a new path and extends it as far as possible, deleting (from G) the edges traversed by this path; procedure Euler_path installs the path sections returned by build_path into the overall Eulerian path that it constructs and returns.

```
PROC Euler_path(G); $ constructs Eulerian path for graph G

IF #(odds := {x IN (nodes := DOMAIN G) | ODD(#G{x}) }) > 2 THEN
```

```
      RETURN OM; $ since more than two nodes are touched by
   END IF;       $ an odd number of edges

   $ Note that -nodes- is the set of all nodes of G,
   $ while -odds- is the set of all nodes of G that are touched by
   $ an odd number of edges

   x := (ARB odds) ? ARB nodes; $ pick a node of -odds- if possible;
                                ^ otherwise pick any node of G

   path := [x] + build_path(x,G);

       (WHILE EXISTS z = path(i) | G{z}/={} )
            path(i..i-1) := build_path(z,G); $ insert new section
                                             $ into path
       END WHILE;

   RETURN path;

   END PROC Euler_path;

   PROC build_path(x,RW G);    $ builds maximal path section starting
                               $ at x, and deletes all edges traversed

   p := [ ];

   (WHILE (y := ARB G{x}) /=OM ) $ while there exists an edge leaving
                                 $ the last point reached

        p WITH := y;                    $ extend path to traverse the
        G -:= {[x,y],[y,x]};            $ edge delete the edge just
traversed
        x := y;                         $ step to y

    END WHILE;

    RETURN p;

    END PROC build_path;
```

11.2 'Topological' sorting

Certain problems, of which scheduling problems are typical, require one to arrange the nodes n of a graph G in a list such that every edge of G goes from a node n1 to a node n2 coming later in the list. This is called the problem of topological sorting. Suppose, for example, that a student must choose the order in which he will take the courses required to qualify as a computer science major, some of which have other courses as prerequisites. Suppose also that we represent the 'prerequisite' relationship as a set G of pairs, agreeing that whenever course n1 is a prerequisite of course n2, we will put the pair [n1,n2] into G. Then, mathematically speaking, G is a graph; in heuristic terms, G{n1} is the set of all courses for which n1 is a pre-requisite. (Note the connection of the 'topological sorting' problem with the transitive computation of prerequsites described in Section

3.3.8.1).

To sort a collection of courses topologically is simply to arrange them in any order in which they could actually be taken, given that all the prerequisites of each course n must be taken before n is taken. To do this is not hard. We simply find some course n which has no (unfulfilled) prerequistes, put n first in the list L, drop all edges [n,nl] from G (since n is no longer an unfulfilled prerequisite) and then continue recursively as long as courses without unfulfilled prerequisites remain. Written as a recursive SETL routine, this is simply

```
(1)     PROC top_sort(G,nodes); $ topological sorting procedure,
                                $ recursive form
        RETURN IF EXISTS n IN nodes - (RANGE G) THEN
          [n] + top_sort(G LESSF n, nodes LESS n) ELSE [ ] END;

        END PROC top_sort;
```

Invocation of top_sort(G) will return a tuple t consisting of some or all of the nodes of G. If it is possible to sort nodes of G topologically, then every node of G will appear in t. This will be the case if and only if G admits no cycle of nodes such that

```
(2)  nl is prerequisite to n2 is prerequisite to n3 is
     prerequisite to... is prerequisite to nk is prerequisite to nl.
```

To see this, note that it is clear that when such a cycle of mutually prerequisite nodes exists, no node in the cycle can ever be put into the tuple t returned by (1). Conversely, if a node n0 belongs to no such cycle then eventually the code (1) will have processed all the predecessors (i.e. prerequisites) of n0, and after this (1) must eventually put n0 into the tuple t it returns. This shows that the set of all nodes belonging to any cycle like (2) is simply

```
    nodes - {x IN top_sort(G,nodes)},
```

so that (1) can also be used to test a graph G for the presence of cycles.

Like many other 'tail' recursions, i.e. recursive procedures which only call themselves immediately before returning, (1) can be rewritten as an iteration (See Section XXX). Written in this way, (1) becomes

```
(3)     PROC top_sort(G) $ first iterative form of topological sort

        nodes := (DOMAIN G) + (RANGE G); $ Here we calculate the set of
        $ nodes; this makes it unnecessary to pass the set of nodes
        $ as an additional parameter.

        t := [ ];                       $ initialize the tuple to be returned

        (WHILE EXISTS n IN nodes - (RANGE G))

            t WITH := n;
            G LESSF := n;
```

```
        nodes LESS := n;

    END WHILE;

    RETURN t;

    END PROC top_sort;
```

It is possible to improve the efficiency of (3) very substantially by keeping the current value of the set (nodes - RANGE G) available at all times. To do this, we proceed as follows:

(a) For each node n, we maintain a count of the number of the predecessors of n which have not yet been put into t.

(b) When n is put into t, we reduce this count by 1 for all nodes nl in G{n}.

(c) If count(x) falls to zero, then x becomes a member of (nodes - RANGE G).

These observations, which could be derived step-by-step from the more general 'formal differencing' principles discussed in Section XXX, underlie to the following revised form of (3):

```
(4)     PROC top_sort(G); $ second iterative form of
                          $ of the toplogical sorting procedure

        nodes := (DOMAIN G) + (RANGE G);

        count := { };     $ initialize the -count- function
                          $ described above

        ready := nodes;   $ The following loop will remove elements
                          $ that have any predecessors from -ready-

        (FOR [x,y] IN G)

          count(y) := (count(y)?0) + 1;
          ready LESS := y;  $ since y has a predecessor
         END FOR;
                            $ At this point -ready- is the set of
                            $ all nodes without predecessors

         t := [ ];          $ t is the tuple being built up

        (WHILE ready/={ })

          n FROM ready;
          t WITH := n;

          (FOR nl IN G{n})
            IF (count(nl) -:= 1) = 0 THEN ready WITH := nl; END;
          END FOR;
```

```
    END WHILE;

    RETURN t;

    END PROC top_sort;
```

It is not hard to see that the preceding code examines each edge of the graph G just twice. Thus the time needed to execute this code is linearly proportional to #G.

11.3 The 'stable assignment' problem

Suppose that the members of a population of n students are applying to a collection of m colleges. We suppose also that each student finds a certain collection of colleges acceptable, and that he/she ranks these colleges in order of decreasing preference. Finally we suppose that each college c can admit only a given quota Q(c) of the students who apply to it, and that it is able to rank all the students in order of decreasing preference. We do not suppose that any of these preferences are necessarily related to any other; that is, different students can rank colleges in radically different orders, and different colleges may find quite different types of students preferable.

The problem we consider is that of assigning students to colleges in such a way as to satisfy the following three conditions:

(a) No college accepts more than Q(c) students;

(b) A college c never admits a student s1 if it has filled its quota Q(c) and there exists an unassigned student s2 to whom college c is acceptable and whom college c prefers to student s1.

(c) There is no situation in which student s1 is assigned to college c1 and student s2 is assigned to college c2, but both the students involved and the colleges involved prefer to switch; that is, s1 prefers c2 to c1, s2 prefers c1 to c2, c1 prefers s2 to s1, c2 prefers s1 to s2.

This problem was studied by David Gale and Lloyd Shapley (American Mathematical Monthly, 1962, pp. 9-15), who gave a simple algorithm for finding an assignment satsifying conditions (a), (b), and (c). The algorithm is just this: Each student applies to his first-choice college. Then each college c puts the topmost- ranked Q(c) students who have applied to it on an active list, and notifies the others that they have been rejected. All rejected students now apply to their second-choice colleges. Then all colleges re-rank their applicants, keep the first Q(c) of these applicants, and again notify the others that they have been rejected. This cycle of re-application and re-ranking continues until no rejected students have any more colleges on their list of acceptable colleges.

It is clear that the assignment produced by this procedure satisfies condition (a). Condition (b) is also satisfied, since if s2 finds college c acceptable, he/she will eventually apply to college c, and can then bump any student s1 whom c finds less acceptable, but will never subsequently be

bumped except by a student whom c finds more acceptable. Finally, condition (c) is satisfied, since if s1 prefers c2 to c1 he/she must have applied to c2 before c1, but been bumped from c2's active list by a student that c2 prefers to s1. But when this happened c2's active list could not have contained any student that c2 does not prefer to s1. Therefore, since the students on college c2's active list never grow any less attractive from c2's point of view, c2 will never regard any student on its final active list as less desirable than s2.

Programmed in SETL, the Gale-Shapley algorithm is as follows.

```
PROC assign(stud_pref,coll_pref,quota); $ Gale_Shapley stable
                                        $ assigment algorithm
     $ we assume that -stud_pref- maps each student into the
     $ vector of colleges he/she finds acceptable, ranked in
     $ decreasing order of preference, and that coll_pref(c)(s1,s2)
     $ is TRUE if college c finds student s1 preferable to
     $ student s2, FALSE otherwise.  The map -quota- is assumed
     $ to send each college into the number of students it will accept.

active := {[c,[ ]]: c IN DOMAIN quota};

     $ set up an empty 'active list' for each college

   applicants := DOMAIN stud_pref; $ initialize the pool of applicants
(FOR j in [1..#quota])             $ we may need as many 'rounds'
                                   $ of applications as there are colleges

    (FOR s IN applicants) active(stud_prefs(j)) WITH:= s; END;

    (FOR c IN DOMAIN active | #active(c) > quota(c))

       (FOR k IN [quota(c) + 1..#active])
         $ drop all 'over quota' applications

         applicants WITH := active(c)(k); $ return student to
                                          $ applicants pool

      END FOR k;

      active(c) := pref_sort(active(c),coll_pref(c));$ re-rank all
                                                     $ who have applied
      active(c) := active(c)(1..#active(c) MIN Quota(c));
                           $ cut back active list

     END FOR c;

     IF (applicants := {s IN applicants | #stud_pref(c) > j}) = { } THEN
       RETURN active;    $ pattern of assignments is complete

     END IF;

   END FOR j;
```

```
END PROC assign;
```

This procedure invokes an auxiliary procedure -pref_sort(t,pref)-, which sorts a tuple t in decreasing order of the pattern of preferences defined by the Boolean-valued map -pref-. We leave it to the reader to develop this procedure. See Exercise 11.11.19 for an additional hint.

11.4 A Text Preparation Program

Text preparation programs aid in the preparation of printed material by arranging text in attractively indented, justified, centered, and titled paragraphs and pages. You may well have used some utility program of this type: they are commonly; available under such names as SCRIPT, RUNOFF, ROFF, etc. In this section, we will describe the internal structure of a somewhat simplified version of such a program.

Our program, which we will call PREPARE, accepts source text containing imbedded command lines as input, and reformats the text in the manner specified by the command lines. Command lines are distinguished from text lines by the fact that the former start with a period as their first character, and by the fact that this initial character is followed by a few other characters signifying one of the allowed PREPARE commands, as listed below. In its ordinary mode of operation, PREPARE collects words from the text it is formatting, and fills up successive lines until no additional words will fit on the line being filled. Then the line is right justified and printed. However, commands can also be used to center a line, and lines can be terminated without being filled (we call this action a 'break'). Text can also be arranged in several special 'table' formats, as described below.

The PREPARE program treats any unbroken sequence of non-blank characters as a word. An 'autoparagraphing' feature, which causes every text line starting with a blank to start a new paragraph, is also available. Margins and spacing are controllable by commands. A '.LIT' command, which causes following text to be printed exactly as it stands, is available to over-ride the normal reformatting action of PREPARE. Facilities for automatic numbering of sections and subsections are also available. If the activity of PREPARE discloses inconsistencies or errors in the commands presented to it, a file of diagnostic warnings is printed.

The formatting commands supported by PREPARE are listed below. However, it will be easier to read these commands if you keep in mind the fact that they sense and manipulate the following variables, which are crucial to PREPARE's activity:

variable name	meaning
Page_horizontal	horizontal width of paper
Page_vertical	number of lines on page
Spacing	current spacing of lines; 1 = single spacing
Left_margin	current indentation for left margin
Right_margin	current right indentation for right margin
Old_margins	saved prior values of margins

Current_line	line of output text currently being built up
Fill	controls collection of words into current_line
Justify	switch controlling right-justification of output lines
Line_count	counts number of lines output so far on current page
Page_number_stack	stack of page and subpage numbers
Number_pages	switch for page numbering
Header_number_stack	stack of section and subsection numbers
Title	current page title
Subtitle	current page subtitle
Chapter_number	current chapter number

The following commands suppported by the PREPARE system are as follows:

.BR (break)

causes a break, i.e. the current line will be output with no justification, and the next word of the source text will be placed at the beginning of the next line.

.S n (skip)

causes a BREAK after which n is multiplied by the number of spaces between lines. The result is the number of lines skipped. Output is advanced to the top of the next page if there is no room on the current page. The parameter n can also have a negative value. Thus, a final footnote can be set by a command such as .SKIP -5.

.B n (blank lines)

causes the current line to be output with no justification, skips n line spaces, and then starts output of the current source text. n can be negative to move the line n lines from the end of the page. BLANK is like SKIP, except that the space to be left is independent of line spacing.

.FG n (figure)

leaves n lines blank to make room for a figure or diagram. If fewer than n lines remain on the current page, text continues to fill this page. Then the page is advanced and n blank lines are left at the top of the next page.

.I n (indent)

causes a BREAK and sets the next line to begin n spaces to the right of the left margin. The parameter n can be negative to allow beginning a line to the left of the left margin. However, a line cannot begin to the left of column 0.

.P n, v, t (paragraph)

causes a BREAK and formats the output paragraphs. The parameter n is optional and, if present, sets the number of spaces the paragraph is to be indented. The default value for n is 5 (n can also have a negative value). v is the vertical spacing between paragraphs. v can range from 0 to 5. (1 denotes single spacing, 2 double spacing, etc.) t causes an automatic TEST PAGE (see the TEST PAGE command).

.C n;text (center)

causes a BREAK and centers the following text in the source file. The centering is over column n/2 independent of the setting of the left and right margins. If n is not given, it is assumed to be the page width.

.NT text (start indented note)

starts an indented note. This command BLANKS 2, reduces both margins by 15, centers the text (if no text is given, it centers the word "NOTE"), and then BLANKS 1. At this point there follows the text of the note.

.EN (end indented note)

terminates a NOTE command, BLANKs 2, and reverts the margins and spacing modes to their settings before the last NOTE command.

.PG (new page)

causes a BREAK and an advance to a new page. If the current page is empty, this command does not advance the page. Just like an automatic page advance, this command adds the title (if given) and page numbers on every page.

.TP n (text page)

causes a BREAK followed by a conditional page advance. It skips to the next page if fewer than n lines are left on the page. This feature serves to ensure that the following n lines are all output on the same page. This command has the form t as an optional argument to the PARAGRAPH command.

.NM n (restart page numbering)

starts page numbering. Pages are normally numbered so there is no reason to issue this command unless page numbering is disengaged. If resumption of page numbering is desired at a certain page, specify n.

.NNM (suspend pagenumbering)

disengages page numbering. However, pages continue to be counted, so that the normal page number can appear if page numbering is re-entered with the NUMBER command.

.CH text (start chapter)

starts a new chapter using the text as the title of the chapter. This command acts as if the following command string were entered:

.BREAK;.PAGE;.BLANK 12;.CENTER CHAPTER n

he n is incremented by 1 automatically. After the CHAPTER n is typed on the page,

.BLANK 2;.CENTER;text;.BLANK 3

occurs. This command then resets the case, margins, spacing, and justify/fill modes. It also clears any subtitles and sets the chapter name as the title.

.NC n (set chapter number)

supplies a number (n) to be used in a subsequent CHAPTER command. NUMBER CHAPTER would be used when a chapter of a document occupies a source file of its own. In such a case, NUMBER CHAPTER would be the first command of the source file.

.T text (define title)

takes the remaining text as the title and outputs it on every page at line 0. The default is no title. If a title is desired, this command must be entered in the source file.

.FT text (define first title)

Same as TITLE, but used to specify the title to be printed on the first page of the document. This command must precede all text in the source file. Use of the FIRST TITLE command is the only way to print a title line on the first page of the document.

.ST text (define subtitle)

takes the remaining text as the subtitle and outputs it on every page. A subtitle appears directly under the page title. The subtitle is not indented, but indentation can be achieved by typing leading spaces.

.SP (start subpage numbering)

executes a PAGE with page numbering suspended. The page

number is unchanged, but letters are appended to the page number. This permits insertion of additional pages within an existing document without changing the existing page numering.

.ESP (end subpage numbering)

disengages the SUBPAGE command by executing a PAGE command with page numbering resumed.

.HD (switch page titling on)

causes the page header (title, subtitle, and page number) to be printed.

.NHD (switch page titling off)

causes the page header (title, subtitle, and page number) to be omitted. The header lines are completely omitted, so that text begins at the top of the page with no top margin.

.J (switch on line justification)

Causes a break and sets subsequent output lines to be justified (initial setting). The command increases the spaces between words until the last word exactly meets the right margin.

.NJ (switch off line justification)

Causes a break and prevents justification of subsequent output lines, allowing a ragged right margin.

.F (switch on line filling)

Causes a break and specifies that subsequent output lines be filled. Sets the justification mode to that specified by the last appearance of JUSTIFY or NOJUSTIFY. FILL adds successive words from the source text until addition of one more word would exceed the right margin, but stops before putting this last word in.

.NF (switch off line filling)

disengages the FILL and JUSTIFY modes. This command is used to permit typing of tables or other manually formatted text.

.LIT (print following text literally)

disengages FILL/JUSTIFY to permit printing of text exactly as entered in source file.

.ELI (end literal text)

used after LITERAL command to re-engage FILL/JUSTIFY.

.LM n (set left margin)

sets the left margin to n. The n must be less than the right margin but not less than 0. The default setting is 0.

.RM n (set right margin)

sets the right margin n. The n must be greater than the left margin. The default setting is 60.

.PS n.m (set page size)

sets the size of the page n lines by m columns. The default setting is 58 by 60.

.SP n (set interline spacing)

sets the number of spaces between lines. The n can range from 1 to 5. The default setting is 1. SPACING 1 is like single spacing on a typewriter and SPACING 2 is like double spacing. SPACING 2 puts one blank line between lines of text.

.AP (switch autoparagraphing on)

causes any blank line or any line starting with a space or tab to be considered as the start of a new paragraph. This command allows normally typed text to be justified without special commands. It does not cause a paragraph if blank lines are followed by a command.

.NAP (switch autoparagraphing off)

disengages the AUTOPARAGRAPH mode.

We now proceed to give SETL code for our text preparation system.

```
Program prepare;                    $ text preparation program

Var                                 $ global variables

  Page_horizontal,                  $ horizontal width of paper
  Page_vertical,                    $ vertical length of paper
  Spacing,                          $ current spacing of lines
  Paragraph_spacing,                $ current spacing between paragraphs

  Left_margin,                      $ current indentation for left margin
  Right_margin,                     $ current right indentation for right margin
  Old_margins,                      $ old margin settings

  Auto_paragraph,                   $ switch which controls 'autoparagraphing'
```

```
      Current_line,              $ line of output currently being built up

      Justify,                   $ controls right-justification of
                                 $ output lines
      Fill,                      $ controls automatic filling of output
                                 $ lines

      Line_count,                $ counts number of lines written so far
                                 $ to current page

      Page_number_stack,         $ stack of page and subpage numbers
      Header_number_stack,       $ stack of section and subsection numbers
      Number_pages,              $ switch for page numbering

      Title,                     $ current page title
      Subtitle,                  $ current page subtitle

      Print_header,              $ switch controlling printing of header
      Chapter_number;            $ current chapter number

CONST Legal_ops = $ Legal commands of PREPARE system
  {BR,S,B,FG,I,P,C,NT,EN,PG,TP,NM,NNM,CH,NC,T,FT,ST,
    SP,ESP,HD,NHD,J,NJ,F,NF,LIT,ELI,LM,RM,PS,SP,AP,NAP};

CONST Cause_new_line= $ commands which advance immediately to a new line
{BR,S,B,I,P,C,NT,EN,PG,TP,CH,J,NJ,F,NF,LIT,ELI,LM,RM};

$ ****** MAIN PROGRAM OF THE TEXT PREPARATION SYSTEM ********

initialize; $ initialize all global variables, determine input and
            $ output files

$ After initialization, we simply enter a loop which adds new words to the
$ current_line as long as input text is available.  All other respect to
$ commands, as well as options such as printing of unfilled
$ lines, printing of text
$ in its literal form, etc. is handled inside the procedure
$ that supplies words to this loop.

(WHILE (word:= next_word( ))/=OM)

       spaces := Spaces_remaining-1;

  IF (Spaces_remaining -:= (#word + 1)) < 2 THEN $ line will be printed

   $ We resort to hyphenation only if there are
   $ less than two words on the current line. In this case, the current
   $ word is hyphenated to fill the current line, and we print as
   $ many lines as necessary to digest the 'word' we now
   $ have, which may be very long.

   IF #Current_Line < 2 THEN $ fill line with piece of word
     Current_Line WITH:= (len(word,spaces) + '-');
   END IF;
```

```
   output(justified(Current_line));

   $ Now we handle possible 'extremely long' words

   (WHILE #word > (Right_margin - Left_margin - 3))

        IF (part:= Len(word,Right_margin - Left_margin))/=OM THEN

              output(part(1..#part - 1) + '-');
              word := part(#part) + word; $ restore first character

    ELSE $ otherwise output the whole remainder of the word

          output(word);
          word := '';

    END IF;

    END WHILE #word;

    $ now we restart the current_line with what remains
    $ of the unpleasantly long word

    Current_line:= IF word = '' THEN [ ] ELSE [word] END;
    Spaces_remaining := Right_margin - Left_margin
                    - IF word = '' THEN 0 ELSE #word + 1 END;

    END IF;                $ i.e., END IF (spaces - remaining etc....

    $ otherwise we just pack one more word into the Current_line.

    Current_line WITH:= word;
    Spaces_remaining -:= (#word + 1);

END WHILE;

    $ We reach this point only when the whole input text has been
    $ processed.  The final line is output, and error messages are
    $ dumped.

    break;

    finalize;

    $ ****** END OF MAIN PROGRAM *****

PROC handle_command(command_tuple); $ command interpreter

$ This command interpreter handles all  PREPARE
$ commands. Like most interpreters, most of its body consists of
$ a single large CASE statement.  Commands will have been pre-validated
$ when this procedure is called

[op,p1,p2]:= command_tuple; $ unpack the command and its parameters
```

```
IF op IN Cause_new_line THEN
  output(unjustified(current_line));  $ output line without justification
END IF;

CASE op OF

       $ First we handle all commands which simply reset
       $ one or more internal global variables of the PREPARE system.

(BR): $ break command

  RETURN; $ nothing more to do for this command

(I): $ indent command

  IF Old_margins = OM THEN
    Old_margins := [Left_margin,Right_margin]; $ save old margins
  ELSE
    [Left_margin,Right_margin] := Old_margins; $ restore old margins
  END IF;

  Left_margin:= (Left_margin + p1) MAX 1 MIN (Right_margin - 10);

(NM): $ resume page numbering

      Number_pages := TRUE;

(NNM): $ suspend page numbering

      Number_pages := FALSE;

(NC): $ Supply chapter number

      Chapter_number := p2;

(T): $ title

     Title := p2; $ set up title

(ST): $ Subtitle

      subtitle := p2; $ set up subtitle

(SP): $ start subpage

      $ This command starts a new (stacked) level of subpage numbering,
      $ allowing subpages to follow pages, etc. without disturbing the
      $ overall prior page numbering

      page; $ output current page
      page_number_stack WITH:= 1; $ start sequence of subnumbers

(ESP): $ end subpage
```

```
        page; $ output current page

        IF #Page_number_stack > 1 THEN $ drop one page level
           junk FROME Page_number_stack;
        END IF;

(HD): $ print page headers

      Print_header := TRUE;

(NHD): $ don't print headers

       Print_header := FALSE;

(J): $ start justification

     Justify := TRUE;

(NJ): $ end justification

      Justify := FALSE;

(F): $ start filling lines

     Fill := TRUE;

(NF): $ stop filling lines

     Fill := FALSE;

(LIT): $ suspend fill and justify

     Fill_j_save := [Fill,Justify]; $ save settings of fill and justify
                                    $ flags
     Fill := Justify := FALSE;

(ELI): $ resume fill and justify

    [Fill,Justify] := Fill_j_save; $ restore previously saved settings

(LM): $ set left margin

    Left_margin := p2 MAX 1 MIN (Right_margin - 10);

(RM): $ set right margin

      Right_margin := p2 MIN Page_horizontal MAX (Left_margin + 10);

(SP): $ set spacing

      Spacing := p2 MAX 1 MIN 5;

(AP): $ start autoparagraphing

      Auto_paragraph := TRUE;
```

```
(NAP): $ End autoparagraphing

       Auto_paragraph := FALSE;

$ Next we handle the few remaining commands which involve
$ some sort of special action

(S): $ skip n lines, with spacing

     blankout((p2 MAX 0) * Spacing);

(B): $ skip n lines, without spacing

     blankout(p2 MAX 0);

(FG): $ leave lines blank for figure, on this page or next
        TO BE SUPPLIED

(P): $ set paragraph spacing
        TO BE SUPPLIED


(C): $ Center text
        TO BE SUPPLIED

(PG): $ Start new page if current page is not empty

      IF Line_count > 1 THEN page; END;

(TP): $ start new page if less than p2 lines remain on current page

      IF Line_count + p2 >= Page_vertical THEN page; END;
        TO BE COMPLETED

END PROC handle_command;

PROC page; $ page advance procedure

$ This procedure puts out a line containing a page advance character,
$ then the page number  the title
$ and the subtitle if these are switched on.  After this, the current
$ line number is re-initialized appropriatedly (automatically, by
$ the action of -output-);

puta(Output_file,Page_advance);

Line_count := 1;

IF Number_pages THEN $ build up first line with page number

    pageno_line := 'PAGE '+/[STR pno + '.': pno = Page_number_stack(i)];
    pageno_line := pageno_line(1..pageno_line-1); $ drop last character
    pageno_line := PAD(    ?????? CMPLETE THIS)
```

```
    output(pageno_line); output('');

END IF;

IF Print_header THEN
    output(Title); output(Subtitle); $ output title and subtitle
    output  $ print blank line
END IF;

END PROC page;

PROC output(line); $ output utility

$ This is the main output procedure of the PREPARE program
$ It sends a line, prefixed by  blank, to the output medium,
$ and then counts up the number of lines sent to the page.  If
$ the page is full a new page is started.  The line is padded out,
$ to give the correct left margin, and over-long output is trimmed.

(FOR j IN [1..spacing])
    puta(Output_file,line);
    line := ' ';  $ print blank line

    Temp_left_margin := Left_margin; $ reset margin in case
                                     $ of autoparagraphing

    IF (Line_count +:= 1) >= Page_vertical THEN page; END;
END FOR;

END PROC output;

PROC break; $ auxiliary end-of-line procedure

IF Current_line = [ ] THEN RETURN; END; $ No output if line empty

output(unjustified(Current_line)); $ output without justification

Current_line := [ ];                $ empty current line

END PROC break;

PROC unjustified(line);             $ converts tuple to string

RETURN ' ' + [wd + ' ': wd IN line];

END proc dont_justify;

PROC initialize;                    $ parameter and file name
                                    $ initialistion routine

$ This procedure initializes all global variables and determines
$ the names of the input and output files.


Page_horizontal := 60; $ default characters per line is 60
```

```
Page_vertical := 58;    $ default lines per page is 58

Spacing := 1;           $ single spacing is default
Paragraph_spacing := 1;  $ Single extra spaces between paragraphs
Left_margin := 5; Right_margin := Page_horizontal - 5; $ default margins

Old_margins := OM;      $ initially, old margins are undefined
Auto_paragraph := Fill := Justify := TRUE;
                        $ all these options are initially switched on

Current_line := '';     $ initially, current line is empty
Page_number_stack := [1];$ initially, on first page
Line - count := 1;        $ start page at first line

Number_pages := TRUE;  $ page numbering switched on is default
Header_number_stack := [ ]; $ initially, no sections or subsections
Title := Subtitle := ''; $ initially no title or subtitle

Print_header := FALSE;  $ headers not printed unless switched on
Chapter_number := 0;    $ will advance with each chapter

Input_file := getspp('PI = PREP.IN/PREP.IN'); $ find input file
Output_file := getspp('PO = PREP.OUT/PREP.OUT'); $ find output file

END PROC initialize;

PROC get_next_line;  $ line reader

$ This procedure reads in the next line of the input file, detects
$ commands, and handles the 'nofill', 'lit', and autoparagraphing
$ features.

get_data: geta(Input_file,line); $ get a line

    $ First we handle The 'nofill' feature.  If -Fill- is false,
    $  we output and clear -Current_line-, either justified or not

IF NOT Fill THEN
    output(IF Justify THEN justified(Current_line)
               ELSE unjustified(Current_line) END);
    Current_line := [ ]; $ clear current line
END IF;

IF EOF THEN RETURN OM; END; $ End of input

IF match(line_copy,'.') /= OM AND
    (cmd := break(line_copy,' ') ? line_copy) IN legal_ops THEN

            ??????   FILL IN

ELSEIF(command_tuple := command_check(cmd,line_copy))/=OM THEN
  handle_command(command_tuple);
END IF;

IF end = 'LIT' AND command_check(cmd,line_copy) /= OM THEN
```

```
    $ We enter literal mode, and output lines
    $ until an .ELI terminator is encountered

   break; $ terminate prior line

   LOOP DO

       geta(Input_file,line);

       IF EOF THEN RETURN OM; END;

       IF match(line, '.ELI' )/=OM THEN QUIT; END;

       output(line);

    GOTO get_data; $ try again for a data line

END IF;

IF Auto_paragraph AND line(1) = ' ' THEN paragraph; END;
                                $ handle  Auto-paragraph feature

spaces := spaces_remaining-1;

RETURN line;

END PROC get_next_line;

    PROC next_word;   $ supplies next word of text

    (UNTIL Input_line = OM)

        IF span(Input_line, ' ') /= OM THEN CONT; END;

        IF (wd := break(Input_line,' ')) /= OM THEN RETURN wd; END;

        IF (line := Input_line) /= ' ' THEN
            Input_line = ' ';
            RETURN line;
        END IF;

        Input_line := get_next_line( ); $ read in next line

    END UNTIL;

    END PROC next_word;

    PROC command_check(cmd,line_copy); $ breaks command out of line

    $ This procedure also checks commands having parameters for
    $ parameter validity, and converts parameters to internal form
    $ where necessary.
    $ In the following map, the symbol I designates an integer, J an
    $ optional integer followed by a semicolon, s a string.
```

```
    CONST parm_descript = $ map defining parameters expected with
                          $ command
        {[S,I],[B,I],[FG,I],[I,I],[P,II],[C,JS],[NT,S],
               [TP,I],[NM,I],[CH,S],[NC,I],[HL,IS],[ST,S],[LM,I],
                      [RM,I],[PS,II],[SP,I]};

    cmd_tup := [cmd];  $ initialize command tuple

    IF (parm_stg := parm_descript(cmd)/= OM THEN

       (FOR p IN parm_stg)
           IF (parm := parm_check(p,line_copy)= OM THEN RETURN OM; END;
           cmd_tup WITH := parm;
        END FOR;

    END IF;

    SPAN(line_copy,' ');  $ span off possible blanks at end of line

    IF line_copy /= ' ' THEN
       error('EXCESS PARAMETERS ON COMMAND_LINE',line_copy);
       RETURN OM;
    END IF;

    RETURN cmd_tup;

    END PROC command_check;

    PROC parm_check(pdes,RW line); $ parameter breakout and check

    $ This procedure breaks a single parameter of a designated kind
    $ out of its -line- parameter, and converts this parameter to
    $ internal form.  If a required parameter is not found, then
    $ OM is returned.  If an optional parameter is not found, then a
    $ default is supplied.

    SPAN(line, ' ');  $ span off initial blanks

    CASE pdes OF $ see the description of check codes in -command_check-

(I): $ required integer

    RETURN integer(line);

(J): $ optional integer, followed by semicolon

    IF (int := integer(line)) = OM THEN RETURN OM; END;

    SPAN(line,' ');

    RETURN IF NOT MATCH(line,';') THEN OM ELSE int END;

(S): $ string. just span off trailing blanks
```

```
RSPAN(line,' ');

RETURN line;

END CASE;

END PROC parm_check;

PROC integer(RW line); $ span off signed integer and converts

IF (parm := SPAN(line,'-0123456789')) = OM
    OR line = '-' OR EXISTS c = line(i) | c = '-' AND i > 1 THEN
      RETURN OM;
END;

RETURN IF MATCH(line,'-')/= OM THEN -VAL line ELSE VAL line END;

END PROC integer;

PROC paragraph;     $ paragraphing procedure

$ This procedure performs a break and then outputs a number of
$ empty lines equal to the current inter_paragraph line spacing.

break;

blank_output(Paragraph_spacing);

END PROC paragraph;

PROC blank_output(nlines); $ outputs blank lines or advances page

$ This procedure outputs -nlines- empty lines if they will
$ fit on the current page.  If not, then the rest of the page
$ is left blank and the page is advanced.
```

(TO BE CONTINUED)

11.5 An Inventory-Control System

In this section we will use SETL to represent a fairly typical commercial application program, namely an inventory control program which could be used to manage inventory and handle a stream of incoming orders for a small-to-medium sized firm. The system will also be responsible for keeping track of customers, shipments, and bills outstanding, for preparation of invoices and dunning letters, and for generation of reports requested by management. To organize commercial applications of this sort is by no means trivial, and our first task is to fix upon a family of concepts which will ease our design task. We begin with two basic notions, namely *data base* and *transaction*. Initially, we will think of a data base simply as a comprehensive collection of maps and other data items used to record the current condition of all objects and records significant to the firm. (Later in this section, however, we will come to a somewhat different and deeper understanding of this concept).

A transaction is a patterned change in the data base which reflects some event or request of which the system must keep track. Transactions can be triggered either exogenously or endogenously. An exogenously triggered transaction begins when input data which describes some external event and gives all relevant parameters of the event is read from a file or from a terminal. In reaction to each such input item, the system must modify the data-base in appropriate fashion, and may also need to generate certain output documents for printing or display.

An endogenously triggered transaction begins either when some specified time limit expires (as in the case of a dunning; letter sent if a bill has remained unpaid for a week), or can be generated internally as a byproduct of some other transaction processed by the system.

The specific example we will consider involves a collection of customers, who order various items supplied by the firm. A certain inventory of each item supplied is held in stock, and as items are shipped these inventories are drawn down. An automatic reorder level is kept on file for each item, and when the inventory of the item falls to this level an order for an additional quantity of the item is immediately issued to the firm which supplies the item.

Customers send in orders, cancellations (of items ordered earlier but not shipped), payments, and notifications of lost shipments. Each customer is extended a certain dollar amount of credit, and as long as the value of items shipped to the customer but not paid for does not exceed this stated amount, additional items are shipped as soon as an order is received. However, if a customer exceeds his credit limit, additional orders will not be shipped. Instead, a letter informing the customer of his delinquency will be generated. Massive over-ordering of this kind will be reported when requested by the firm's financial management.

When a valid order is received and processed, a shipment order is generated for all the items which the original order lists. This shipment order goes to the firm's warehouse, which attempts to crate and ship the items requested. Once this has been is done, the warehouse sends a shipment confirmation notice for the items which it has been possible to ship, the material shipped becomes billable to the customer, and an invoice is issued. If payment is made within 60 days, no additional charge is due. Otherwise a surcharge of 1% per month on unpaid balances is levied. A monthly bill representing unpaid charges is sent. Orders transmitted to the warehouse but not promptly confirmed generate follow-up messages and, eventually, apologies to the customer.

Customers can report non-delivery of invoiced items, can return items shipped and be credited for the value of these items, and can cancel orders for items that have not been shipped.

When items ordered from suppliers are received by the warehouse, the warehouse issues a delivery notification slip noting the arrival of these items; this may allow various suspended orders to move forward. The warehouse can also issue spoilage reports for given items. Finally, after taking physical inventory, the warehouse can indicate the quantity of each item that is actually on hand.

Every transaction entered into the system is issued a transaction number which the key entry operator who enters the transaction can copy onto the sheet of paper with which the transaction originates. Transactions submitted to but rejected by the system are logged in a rejected transactions log, which can be printed or examined later to determine the reason for the rejection, e.g. mis-keyed data, illegal parameters, etc.

Clear and error-free implementation of this whole list of transactions may at first glance appear to be a forbidding task. To reduce the difficulty of an initial attack on the design problem we face in creating the required system, we will make use of a powerful programming tool, namely the parallel-process extensions to SETL described in Section XXX. Why is this appropriate? We can answer this question as follows. Our inventory control application, like other commercial application of the same general sort, can be regarded as an 'event tracker'. That is, it aims to follow an evolving sequence of real-world events (each transaction processed by the system notifies it of one such event), and to maintain a model of the real-world situation generated by those events. As significant events come to its attention, the system is also expected to respond appropriately (e.g. by issuing bills and dunning letters, noting payments, etc.) Our inventory control system, like all other such systems, must also aim to detect and report various kinds of anomalous situations, which either reflect 'exceptional' real-world occurrences, for example non-delivery of an item ordered from a supplier, or which reflect the fact that the inventory control program's logic, no matter how carefully worked out, cannot be a complete representaion of all possible real-world event sequences.

If we begin by trying to work out a centralized, single-thread program capable of dealing with all the events and situations which can occur in an application of this sort, confusion can easily result. The fact that some of the consequences of a transaction only take place hours or days after processing of the transactions begins is troublesome. Equally troublesome is the fact that certain transactions, e.g. transmission of a shipping order to the warehouse, saddle us with the responsibility of checking periodically for a follow-up transaction that we must expect, namely confirmation of shipment or notification of a out-of-stock condition.

The programming structure necessary to manage all this is more easily comprehended if we decentralize our approach to it. This means that we will want to think, not in terms of one central record keeper which must keep track of everything, but in terms of multiple processes acting in parallel, each of which is responsible for overseeing a single, narrow activity, essentially a single transaction and its delayed consequences, from start to finish. Using this idea, we use it vigorously: for example, we introduce a seperate process for each customer, and also for each supplier, each order, and each item supplied. We therefore program as if the firm whose operations we are modeling assigned responsibility for all communication with a particular customer or supplier to a single (mechanized) account representative, who has this responsibility and no other. In the same sense, we program as if the firm hired a new junior clerk to process every single order received, giving him responsibility only for this one order, and discharging him as soon as the order is either shipped or abandoned as unfillable.

Although it is true that these processes will have to interact, enough of what needs to be done is interaction-free for the introduction of these many parallel processes to simplify our approach considerably. Were this not the case, large businesses could never have developed, since in reality they do and must make use of numerous employees with simple narrow responsibilities and authority, who can in fact transact large volumes of the kind of business necessary without being overwhelmed by any growing requirement to interact.

In the first programming approach shown below, we give each process a responsibility narrow enough for the nature of its activities to remain clear. Where interactions complex enough to become troublesome threaten to appear, we avoid them by having processes request appropriate services from other, logically somewhat more central, processes. Generation of overall summaries of system condition is accomplished by having centrally placed processes communicate requests for information to the less centrally placed processes which they are responsible for managing.

From this decentralized point of view, a commercial data base simply records the condition of the numerous parallel processes which exist (mostly in a state of 'suspended animation') within the world of processes collectively constituting the application, plus a few sets and maps which hold necessary global information, and which also serve to correlate processes with the particular activities for which they are responsible. In the system that we wish to program, the following transactions will be supported (capital letters in brackets show transaction codes).

Open account (OA)	start a new customer account. A unique customer identifier, an address string, and credit_limit must be supplied.
List accounts (LA)	print sorted list of all accounts, with customer address, credit_limit, credit_remaining, and last_payment_date
Modify address (MA)	revise customer address/telephone number string
Modify credit limit (MCL)	revise customer credit limit
Close account (CA)	delete account
Record payment (P)	note receipt of payment from customer
Record return (R)	note customer return of portion of previous shipment
Record cancellation (C)	note customer cancellation of unshipped portion of order
Note loss of shipment	note that shipment has disappeared in transit, and make necessary adjustments

(NL)

Order from customer (O)	note arrival of a new order, check on existence of corresponding customer and on credit availability
Note receipt of stock (RS)	note arrival of specified item, in quantity k, from its supplier. This transaction will generate a receipt to be transmitted to the supplier, update warehouse inventories, etc.
Note item spoilage (NSP)	adjust recorded inventory of given item to reflect spoilage reported by warehouse.
Note current inventory (NI)	note inventory as recorded by warehouse
Begin stocking item (BI)	enter a new item into the catalog of items managed by the system. A unique item identifier, supplier, price, reorder_level, reorder_amount, and initial item stock must be specified.
End stocking item (EI)	delete item from catalog of items supplied
Change item information (CSI)	change item supplier, price, reorder-level, and reorder amount
Enter transaction authorization (E au)	the name of a transaction and an employee number are supplied. This employee becomes capable of authorizing transactions of a specified kind.
End transaction authorization (KAU)	the name of a transaction and an employee number are supplied. The employee's right to authorize transactions of a given kind is cancelled.
Print rejected transaction file (PRJ)	output function: print the file of all transactions rejected as illegal since the last time that this output operation was performed.
Print customer summary (PCS)	print status of specified customer, including pending orders, and dates and totals of last few shipments.
Print volume summary (PVS)	print total value of shipments, alphabetized by customer, for specified period of days prior to present.
Print supplier (PS)	print status of specified supplier.

Print comprehensive summary of suppliers (PAS) — print alphabetized catalog of all suppliers.

(MORE SUPPLIER-RELATED TRANSACTIONS ARE NEEDED)

Having now described the commercial application that we mean to program, we go on to describe the processes which we will use to realize it. These are as follows:

(a) For each valid customer, a customer process is created. This keeps track of such basic customer-related information as address, telephone number, etc., tracks the customer's current credit balance, credit-limit, etc. and manages all lower-level processes acting on behalf of the customer.

(b) For each valid order, an order process is created. This keeps track of the shipment of the order (part of which may be delayed), and notifies the relevant customer process of the date and amount of shipmments. Order cancellations are handled by these order processes. Order processes held up unduly long will send out letters of apology.

(c) For each item supplied, an item process. This keeps track of the inventory of the item, reorder level, dates and amounts of reorder, expected date of arrival of new supplies. Item spoilage is also noted by this process.

(d) A master process reads an input file of transactions and routes each transaction to the appropriate lower level process. Transactions that cannot be handled are posted to a rejected transactions file.

(e) For each employee, an employee process. This sends each employee a periodic report concerning all significant transactions which the employee has been reported as authorizing.

(f) For each supplier, a supplier process, which makes up orders for transmittal to the supplier and prints these orders daily. The code for the inventory-control system described in the preceeding pages begins here.

```
PROC master;  $ the master process

    LOOP DO

        AWAIT (transaction /= OM);    $ we assume for simplicity that the
                                      $ transaction is given as a tuple

        IF (p := proc_handling(transaction)) = OM THEN CONTINUE; END;
            $ bypass transactions failing initial check

        CASE tc := transaction(1) OF $ tc is the transaction code

    (OA):   $ open a new account

     result := new_account(transaction);     $ 'result' may report an
```

```
                                        $ error checked for below.

(LA):  $ list all accounts

    report_vector := [i[rpt := report(p),id],rpt(1)]:
                          p = customer_process(id)];
    print_report(sort(report_vector));

(MA):  $ modify address

    p.address := transaction(4);  $ set address value of customer
                                  $ process

(MCL): $ modify credit limit

    p.cred_lim := transaction(4); $ set credit limit value of customer
                                  $ process

(CA):  $ close account

       $ This rather delicate instruction is transmitted to a
       $ appropriate user process,  which proceeds to commit
       $ suicide in an orderly fashion.

       p.work WITH := DIE;  $ instruction to close up shop
       proc_of(transaction(XACCOUNT)) := OM;
              $ eliminate account number

(P,R,C,NL,O):  $ payment, return, cancellation, loss of shipment,
               $ order:  handled by customer processes
       p.work WITH:= transaction;

(RS): $ note receipt of stock: handled by supplier process

          IF (p := supplier(transaction(2))) = OM THEN $ item not
                                                       $ stocked
          print_return_order(transaction);  $ signal warehouse to
                                            $ return shipment
          ELSE
               p.work WITH := transaction;
          END IF;

(EI): $ delete item from catalog of items supplied

      item_of(transaction(XITEM) := OM; $ remove from catalog
      p.work WITH := DIE;  $ send termination command

(CSI): $ change supplier for item

                          (TO BE SUPPLIED)

(NSP,NI): $ take note of spoilage reported by warehouse,
          $ note actual inventory

      $ These are passed along to the item processor
```

```
      p.work WITH := transaction;

(BI):     $ begin stocking item

     $ This begins by checking the validity of the indicated
     $ supplier.  If invalid, the transaction is rejected;
     $ Otherwise an item proscess is created; the supplier,
     $ price, reorder_level,reorder_amount, and initial item
     $ stock are passed to this process

                              (TO BE SUPPLIED)

(EAU): $ enter transaction authorization

       authorizations WITH := [transaction(Xauth + 1..Xauth + 2),
                               transaction(Xauth)];

(KAU): $ end transaction authorization

       authorizations LESSF := transaction(Xauth + 1..Xauth + 2);

(PRJ): $ print rejected transaction file

       print_rejected(bad_transacts);
       bad_transacts := [ ];  $ restart file

(PC): $ print status of specified customer: handled by;
      $ customer process

      p.work WITH := PC;

(PVS): $ print total value of shipments, alphabetized by
       $ customer for specified period of days prior to present

                        (TO BE SUPPLIED)

   END CASE;

   END LOOP;

END PROC master;

PROC customer; $ the customer process

LOOP DO

    AWAIT work /= [ ]; $ wait for work to turn up
    transaction FROMB work;

    CASE tc := transaction(1) OF

(P): $ note receipt of payment

     balance +:= transaction(4); $ increment the recorded balance
```

```
(R): $ note customer return of part of previous shipment
     $ all items returned must be credited and added to inventory

     (FOR [item,no] IN transacton(4..))

        IF(ip := item_proc(item))/=OM THEN
        balance +:= ip.cost * no; $ increment recorded balance
        ip.work WITH := [R,no]; $ post to ip for inventory revision
        ELSE $ return of item no longer stocked

        post_bad([transaction,item,'NOT STOCKED');

        END IF;

        END FOR;

(NL): $ note customer claim of non-delivery

      $ here we check total amount of non-delivery claimed this
      $ year, comparing it to dollar volume shipped.  If this
      $ exceeds 1%, extra authorization is required to accept
      $ the transaction

      [total,not_stocked] := total_of(transaction);

      IF (new_nondel := non - delivered_value + total) * 100
                                          > volume_shipped
          AND NOT strong_authorization(transaction(2)) THEN $ reject
         post_bad([transaction,0],'EXCESSIVE NON-DELIVERY');
      ELSE
         non_delivered_value := new[nondel;
         balance +:= total; $ credit customer for return
         post_bad([transaction,not_stocked],'NOT STOCKED');
      END IF;
(O): $ process an order

     $ We verify that the orders pending plus the balance
     $ outstanding do not exceed the customer's credit limit.
     $ If this condition is met, we create an order processor to
     $ handle the order; if violated, the order is simply handed
     $ on for retry.
     [total,not_stocked] := total_of(transaction);
     IF (orders_pending + balance+total) <= credit_limit THEN
        order_pending +:= total;
        orders_going WITH := NEW order(transaction, total,
                                       not-stocked,SELF);
     ELSE
        retryer.work WITH := transaction;
     END IF;

(S): $ process shipment

     $ Shipment notification consists of the order_number items
     $ shipped quantitites, and prices.  This transaction is
```

```
        $ executed once each day.

        print_invoice(address,shipment_list);

                            (TO BE CONTINUED)

 PROC order(transaction,estimated_total,not_stocked,cust_proc);

       $ The order process

       $ This process oversees a single order until it has
       $ either been fully shipped, or cancelled,
       $ or until it has been posted as bad because of
       $ lack of response from the warehouse

                        (TO BE CONTINUED)

PROC total_of(transaction); $ auxiliary process to check a
                            $ list of items ordered for validity

total := 0; $ total cost of items ordered
not_stocked := [ ] $ items not stocked

(FOR [item,no] IN transaction(first_data))

    IF(ip := item_proc(item)) /= OM THEN
        total := ip.cost * no; $ increment total
    ELSE
        not_stocked WITH := item;
    END IF;

END FOR;

RETURN [total,not_stocked];

END PROC total_of;
```

Since we have used multi-process primitives to represent the inventor control application shown in the preceding pages, our approach to thi application makes use of programming-language facilities that are not ordnarily available. To remove the sting from this objection, we will no reprogram the application in a manner which avoids the use of paralle process facilities and can be used in an ordinary 'batch' environment. In this alternative representation, a data-base in the ordinary sense, that is, a family DB of sets and maps, is used to represent what would otherwise b the collection of states of all processes active in the system at any give moment. When possible, we call a procedure to accomplish an action immediately rather than some subsidiary process to perform the actio Where this is not possible, we post the necessary request to a workpile pending requests, which is digested when no more pressing work remains . Requests to inititate an activity at a later time are simply written out to a final master file which the system creates at the end of each batch ru This file consists of two parts, a first being the collection of accumulate requests to initate activities during later runs, and the second being a

full copy of the data-base DB. Use of this file gives our second version of the inventory control system the processing pattern the manner typical for commercial batch processing, namely run of the system applies a file, of transactions thereby a master file thereby producing an updated master file.

11.6 A Turing-Machine Simulator

Turing machines, named after the famous English mathematician and computer scientist Alan Turing, are the most elementry kind of computer; so elementary that they are not used in any practical way, but merely serve as idealized models of computation at its simplest. Used in this way, they play an important role in theoretical investigations of the ultimate limits of computability. A significant fact about these very simple computing mechanisms is that they can be programmed to imitate the action of any other computer, for example, a Turing machine can be programmed to take the text of any SETL program and print out its result.

Turing machines consist of two basic parts: a tape and an read-write head. The tape is simply a linear array of squares, infinite in both directions. In a tape square, the automation can print any character chosen from a finite collection called the tape alphabet of the Turing machine. All but a finite number of squares on the tape are always blank. At the start of each cycle of operation of the Turing machine, its read-write head is positioned at one of the tape squares, and is in one of a finite collection of possible internal states s. The read-write head then reads the character c held in the square at which it is positioned and performs three actions, all determined by the character c which has just been read and the internal state s of the read-write head:

(i) some new character c′ is written into the tape square at which the read-write head is positioned, replacing the character c that was there;

(ii) the read-write head passes into a new internal state s′;

(iii) The read-write head moves either one step right, one step left, or remain where it is.

Plainly, these actions of the Turing machine can be defined by a map action(c,s), whose two parameters are a tape character c and an internal state s, and whose value is a tuple [c′,s′,n′], consisting of the tape character c′ that will over-write c, the new internal state s′ of the read-write head, and an indicator n of the direction of head motion, which must be either +1 (move right), -1 (move left), or 0 (don′t move).

The following procedures read in the description of a Turing machine, check this description for validity, read in the initial contents of the Turing machines′s tape, and then proceed to imitate its actions. The tape is represented by a tuple -tape- whose j-th component is the character written in the j-th square. Blank squares contain the blank character. The Turing machine stops when it reaches an internal state s such that action(c,s) is undefined. We assume that the Turing machine description read in initially is a set of quintuples [c,s,c′,s′,n′], each representing an action- map entry [[c,s],c′,s′,n]. This description is checked to verify

that the action map it describes is really single-valued. The auxiliary procedure -print_tape- prints the contents of the Turing machine tape after each cycle of operation.

```
PROGRAM Turing_simulate;  $ Turing machine simulator

IF (atps := read_check( )) = OM THEN RETURN; END;
                          $ illegal specification

[action, tape, position, state] := atps;
  $ unpack action table, initial tape, initial position, and
  $ initial state

(WHILE (act := action(tape(position),state)) /= OM) $ until stop

     [tape(position),state,n] := act; $ write new character to tape,
                                      $ and change internal state

     IF (position +:= n) < 1 THEN     $ moved left to brand-new square
       tape := [' '] + tape;          $ add blank square at left
       position := 1;                 $ and adjust position pointer
     ELSEIF position  > #tape THEN    $ moved right to brand-new square;
       tape WITH:= ' ';               $ add blank at right
     END IF

     print_tape(tape,position);

  END WHILE;

  print('Simulation ended. Character and state are:',
                           tape(position),state);

  END PROC Turing_simulate;

  PROC read_check;  $ reads and checks action table, tape,
                    $ initial position, and initial state

 MACRO check(condition,message,quantity); $ utility macro for
                                          $ input-condition checks
 IF NOT condition THEN
   print(message,quantity); $ print diagnostic message and offending
                            $ quantity
   RETURN OM;  $ as indication of error

 END IF;
 ENDM;

  read(actuples,tape,position,state);
  action := {[[c,s],c2,s2,n]:[c,s,c2,s2,n] IN actuples};

  (FOR im = action{cs} | #im > 1) $ action is not single valued

    print;
    print('action is indeterminate in condition',cs);
    print('actions could be:');
```

```
        (FOR [c2,s2,n] IN im)
           print(c2,s2,n);
        END FOR;

        print;
      END FOR;

      RETURN OM; $ as indication of error in action table

     check((bad_cs := {cs: [c2,s2,n] = action(cs)
                        | n NOTIN {-1,1,0}}) = { },
      'Illegal tape-motion indicators occur for conditions:',bad_cs);

     check(is_integer(position),'Illegal initial position:',position);
     check(is_tuple(tape),'Illegal inital tape:',tape);
     check(FORALL t=tape(i) | IS_STRING(t) AND #t=1,
                              'Illegal initial tape', tape);

$ now add extra blanks to the initial tape if necessary

     IF position > #tape THEN  $ extend tape with additional blank squares
        tape +:= (#tape - position) * [' '];
     ELSEIF position < 1 THEN $ add extra blank squares to left
         tape := (1 - position) * [' '];
         position := 1; $ adjust index of position on extended tape
     END IF;

     RETURN [action,tape,position,state];

     END PROC read-check;

     PROC print_tape(tape,position); $ Turing machine tape print utility.

     $ This procedure is used to display the state of the Turing machine
     $ tape at the end of each cycle of simulation

CONST  sq=18, hsq=9;          $ one fourth and one eigth screen size
CONST  screen_size = 72;      $ number of characters on terminal
  topline := screen_size * ''; topline (4*hsq+1...4*hsq+4):='****';
  botline := screen_size * '-';
  tape_string := hsq * ' ' +/ tape + hsq * ' ';
       $ Convert tape to string and pad with blanks.
  tape_string := tape_string(position - hsq.. position + hsq-1);
  picture := +/['1 ' + t + ' ' : t IN tape_string];
  picture(1) := ' ';          $ Remove first vetical bar.
  print; print(topline); print(picture); print(botline);

     END PROC print_tape;

END PROG Turing_simulate;
```

11.7 'Huffman Coding' of Text Files

The standard 'ASCII' alphabet of computer characters contains 256 characters, each of which is represented at the internal machine level (see Section 9.3.2) by a sequence of 8 binary 'bits' (i.e., zeroes and ones). If large volumes of English-language text need to be stored, this internal coding, which uses just as much computer memory space to represent a rare character like 'Z' as to represent a common character like 'e', is by no means optimal. It is better to represent frequently occuring characters by shorter sequences of bits, even though this forces one to lengthen the internal encoding of less frequent characters, since overall this will diminish the total storage reqired to store typical texts. An effective method for using 'variable length' encodings of this kind was described by X. Huffman and has become known as Huffman coding. Huffman's technique is to arrange all the characters to be encoded as the terminal nodes of a binary tree, in the manner shown in Fig. 1. This tree should be set up so that commonly occuring characters appear near its 'root' node and rare characters appear at a greater distance from its root.

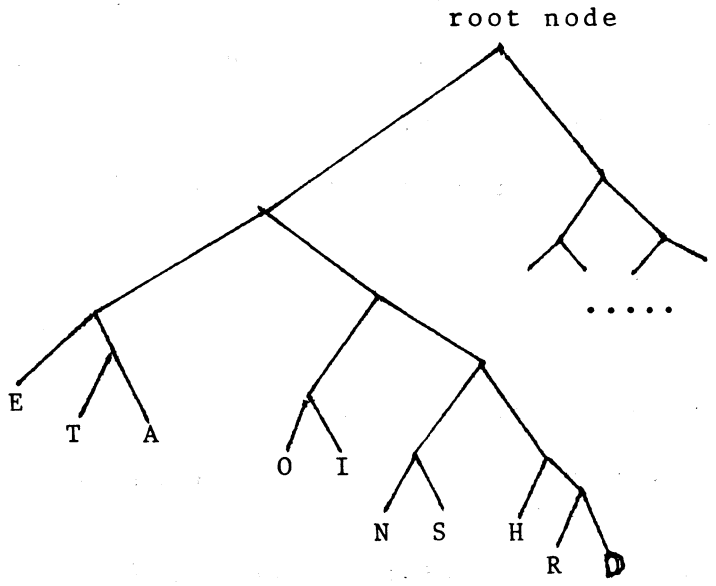


Fig. 1 Binary 'Huffman tree' with characters attached to its terminal nodes.

There will always exist a unique path from the root node of such a tree to each terminal node or 'twig' of the tree, and any such path can always be described by a unique sequence of zeroes and ones, where '0' means 'take the left branch' and '1' mean take the right branch down the tree. As the code for a character c we can therefore use the binary sequence describing the path from the root node of the tree to the terminal node at which c is attached. For example, the tree shown in Fig. 1 would assign the code '000' to 'E', the code '0010' to 'T', the code '0101' to I, etc. To encode a sequence of characters, we simply concatenate the sequences of zeroes and ones representing its individual characters. To decode a sequence s of zeroes and ones, we start from the root of the Huffman tree which defines our encoding, and use the leftmost bits of s to guide us down a path in the tree. As soon as we reach a twig of the tree we add the character attached to this twig to the sequence of decoded characters we are building up. The

sequence of bits that led us to this character is then detached from s, and we return to the root of the Huffman tree and continue the decoding process using what remains of s.

The three routines which follow embody this encoding and decoding technique. The -Huff- procedure takes a character string and encodes it using Huffman's method. -Puff-, which is the inverse of -Huff-, takes the encoded form of a string s and recovers the original form of s. The third procedure, called -setup- , takes maps -lef-t and -right- representing a Huffman tree and uses them to initialize various global data objects required by the -Huff- and -Puff- routines.

```
MODULE Huffman - Huff_Puff; $ Huffman encode, decode, and setup
VAR
    H_code,  $ maps each character into its Huffman code
    H_root,  $ root node of Huffman tree
    H_left,  $ maps each node of the Huffman tree to its left
             $ descendant
    H_right, $ maps each node of the Huffman tree to its right
             $ descendent
    H_char;  $ maps terminal nodes of the Huffman tree to the
             $ characters they represent

PROC setup(root,left,right, char); $ auxiliary initialization routine
    $ We begin by using the procedure arguments to initialize
    $ all but the first of the global variables listed above.

H_left := left; H_right := right;
H_root := root; H_char := char;

$ Next we calculate H_code(c) for each character c

parent := {[y,x]: [x,y] IN (H_left + H_right)};
    $ This maps each tree node to its parent

H_code := { };  $ begin calculating Huffman codes from tree structure

(FOR c = H_char(node))

 $ chain up to the root, noting how we got there

   bits := ' ';  $ initially, path is null

   (WHILE node /= H_root)
       bits := IF H_left(par := parent(node)) = node THEN
                          '0' ELSE '1' END + bits;
       node := par;  $ step up to parent
   END WHILE;

   H_code(c) := bits; $ record Huffman code for current character

END FOR;

END PROC setup;
```

```
PROC Huff(stg);     $ calculates Huffman code for string -stg-

RETURN '' + /[H_code(c): c = stg(i)];
                    $ concatenate codes of individual characters

END PROC Huff;

PROC Puff(Huff_stg); $ decodes a Huffman-coded string

stg := '';          $ initialize decoded string
node := H_root;     $ start at Huffman-tree root

(FOR b = Huff_stg(j)) $ examine binary bits of Huff_stg in order

    node : = IF b = '0' THEN H_left(node) ELSE H_right(node) END;

    IF(c := H_char(node))/= OM THEN $ have reached twig

      stg +:= c;       $ append to decoded portion
      node := H_root; $ restart at Huffman-tree root

    END IF;

END FOR;

RETURN stg;

END PROC Puff;

END MODULE;
```

The encoding and decoding procedures shown above sidestep the question of how to find the tree that will give us a maximum degree of text compression. Of course, the rule for finding this tree, given the frequency with which each character occurs in the text we are to encode, is Huffman's essential discovery. His rule is as follows: we begin by finding the two characters c1, c2 of lowest frequency. These are then logically 'conglomerated' into a single joint character c, of which c1 and c2 become the left and right descendants respectively. We remove c1 and c2 from the collection of characters which remain to be processed, and replace them by c. Continuing this until only one character remains, we will have built the Huffman tree.

Represented in SETL, this procedure is as follows:

```
PROC Huff_tree(freq); $ Huffman tree-build routine

  $ -freq- is assumed to map all the characters of our alphabet
  $ into their expected frequencies of occurrence.

  $ This procedure returns a quadruple [root,left,right,char]
  $ consisting of the Huffman tree root, its left and right
  $ descendancy maps, and a map -char- which sends each terminal
  $ node of the tree into the character attached to this node.
```

```
    $ Since the code which follows will represent tree nodes by
    $ character strings, the -char- map is just the identity map on
    $ single-character strings, and is conveniently set up right here.

  char := {[c,c] : c IN DOMAIN freq};

  left := right := { };  $ initialize the descendancy mappings

  (WHILE #freq > 1)

    [c1, freq_c1] := get_min(freq); [c2,freq_c2] := get_min(freq);
    freq (c := (c1 + c2)) := freq_c1 + freq_c2;
         $ form a logically 'concatenated' character
    left(c) := c1; right(c) := c2; $ make c1 and c2
                                   $ descendats of c

  END WHILE;

  RETURN [ARB DOMAIN freq, $ which is necessarily the tree root
                  left, right, char];

  END PROC Huff_tree;

  PROC get_min(RW freq);
    $ This auxiliary procedure finds the character c of minimum
    $ frequency, returns c and its frequency, and deletes c from
    $ the domain of -freq-.  Note that it uses a 'dangerous' program
    $ construction, legal in SETL, but certainly not recommended
    $ for use in any context which is at all complex, namely it is
    $ a function which modifies the argument with which it is called.

    min_freq := MIN/[f: f = freq(c)];

    ASSERT EXISTS f = freq(c) | f = min_freq;

    freq(c) := OM; $ modify the input argument (which is 'RW').
                   $ DANGEROUS!

    RETURN [c,f];

    END PROC get_min;
```

Various improvements and extensions of the procedures described in this section appear in Exercises 13-18.

11.8 A 'Game Playing' Program

In this section, we will explore the basic structure of programs which play board games, like chess or checkers which involve two players, whom we shall call 'A' and 'B'. The momentary state s = [p,x] of any such game can be defined by giving the position p of the various pieces or counters used in the game, and by stating which of the players, x = 'A' or x = 'B', is to

move next. Given any such state s, the rules of the game will determine the moves which are legal, and hence will determine the set of all possible new states s1,...,sk, exactly one of which must be chosen by the 'active' player, i.e. the player whose turn it is to move. We shall suppose in what follows that the map has_turn(s) determines this player (i.e. has_turn(s) is just x, if as above s has the form [p,x]). We also suppose that the map next_states(s) gives us the set {s1,..sk} of states to which the active player can move.

Any such game will end as soon as certain states, called terminal states, are reached. (In chess, for example, these are the states in which one of the players has been 'checkmated'). For purposes of analysis it is convenient to suppose that when a terminal state s is reached, D dollars are transfered from player B to player A. We can either suppose that the sum D is fixed, or that it depends on s. It is actually more convenient to make the latter asumption, and we shall do so, supposing accordingly that we are given a function A_wins(s) defined on all terminal states s, and that when a terminal state s is reached the sum = A_wins(s) is transfered from B to A. ; Plainly A is the winner if D > 0, B is the winner if D < 0, and the game counts as a tie if D = 0. It is convenient to suppose that A_wins(s) = 0M if the state s is not a terminal state; then the condition A_wins(s)/=0M can be used to test for terminal states.

The three functions, has_turn(s) (whose value must be either 'A' or 'B'), next_states(s), and A_wins(s) serve to encapsulate the basic rules of any two-player game we wish to study.

Next, to begin to understand the strategic considerations which determine the laws of effective play, it is useful to extend the function A_wins(s), which is only defined for terminal states, so that it becomes a function A_can_win(s), defined for all states. We do this in the following recursive way:

```
(1)     A_can_win(s) = A_wins(s)?
        IF has_turn(s) = 'A' THEN MAX/[A_can_win(sy):sy IN next_states(s)]
                       ELSE MIN/[A_can_win(sy):sy IN next_states(s)] END;
```

The meaning of this formula can be explained as follows:

(a) If the state s is terminal, the game is over and the amount that A can win is exactly the smount that A has in fact won.

(b) Otherwise, if it is A's turn to move , he will chose the move that is most favorable to him, shifting the game into that state sy in next_states(s) for which A_can_win(sy) is as large as possible. Conversely, if it is B's turn to move, he will defend himself as well as possible against A's attempts to win a maximum amount. B does this by shifting the game into the state sy for which A's attainable winnings are as small as possible. Since A wins what B loses, and vice-versa, this is at the same time the state in which B's winnings are as large as possible.

It is not hard to see that if the function A_can_win defined by (1) is known, and if both players expect their opponents to play with perfect accuracy, player A should always use his turn to move to a state sy such that A_can_win(sy) is as large as possible, and player B should always use

his turn to move to a state sy such that A_can_win(sy) is as small as possible. To show this, suppose that the sequence of states traversed in the history of a game, from the moment at which it reaches state s, up to the moment at which the game terminates, is s = s1,s2,...,sn. Using (1) it is easy to see that if A uses this strategy, A_can_win(sj) will never decrease, so that by using our recommended strategy A guarantees that when the game terminates he will win at least the amount A_can_win(s). Conversely, if B uses the strategy we recommend, then formula (1) shows that A_can_win(sj) will never increase. Hence, if player A ever makes a move which decreases the value of A_can_win from v to some value u which is less than v, then after this B can prevent him from recovering, i.e. from ever winning more than U. If follows that, if A gives his opponent credit for playing optimally, A must never 'give ground' in regard to the function A_can_win(s), i.e. that when it is his turn to move he should always move to a new state sy such that such that A_can_win(sy) is as large as possible. (Of course, if he does this, then A_can_win(sy) = A_can_win(s); see (1)).

Reasoning by symmetry, we also see that B should always move to a new state sy such that A_can_win(sy) is as small as possible.

These considerations indicate that any game-playing program will need to calculate the function (1). However, if the game being analysed is at all complex, it will not be feasible just to use the recursive definition (1) as it stands, since the tree of possible moves and countermoves which (1) would examine will tend to grow very rapidly. For example, if at every level A has just 4 possible moves and B has 4 possible countermoves, then 256 different positions can evolve from an initial state s after A and B make two moves each, 64,000 different postions after A and B have made 4 moves each, and hence the recursion (1) would have roughly 16,000,000 positions to examine if we used it to look ahead through all possible combinations of six moves of A and six countermoves of B.

This makes it plain that it is important to accelerate calculation of the function A_can_win(s) as much as we can. Several techniques for doing this have been developed, but we shall only describe one particularly important method of this kind, the so-called 'alpha-beta pruning' method. To derive this improvement, suppose that f is a function mapping numbers to numbers, and that f is monotone, i.e. has the property that x <= y implies f(x) <= f(y). Then since x MAX y is the larger of x and y, it follows that f(x MAX y) = f(x) MAX f(y). Hence

(2) `f(MAX/[e(x):x IN s]) = MAX/[f(e(x)):x IN s]`

for every set s and expression e. It is also clear that (2) continues to hold if we replace MAX by MIN. This remark, and (1), make it obvious that the following recursive function calculates the function B(s,lo,hi) = A_can_win(s) MIN hi MAX lo:

```
(3)     PROC B(s,lo,hi);

        IF(v := A_wins(s)) /= OM THEN RETURN v MIN hi MAX lo; END;

        IF has_turn = 'A' THEN

          max_till_now := lo;
```

```
      (FOR sy IN next_states(s))
        max_till_now MAX := B(s,lo,hi);
      END FOR;

      RETURN max_till_now MIN hi;

    ELSE

      min_till_now := hi;

      (FOR sy IN next_states(s))
        min_till_now MIN := B(s,lo,hi);
      END FOR;

      RETURN min_till_now MAX lo;

  END IF;

  END PROC B;
```

Since the quantity returned at the end of the first loop in (3) is max_till_now MIN hi, we can terminate the loop as soon as max_till_now rises to -hi-; and a similar remark clearly applies to the second loop in (3). This crucial observation allows us to rewrite (3) as

(4)

```
  PROC B(s,lo,hi);

  IF(v := A_wins(s)) /= OM THEN RETURN v MIN hi MAX lo; END;

  IF has_turn = 'A' THEN

    max_till_now := lo;

    (FOR sy IN next_states(s))
      IF(max_till_now MAX := B(sy,lo,hi)) >= hi THEN RETURN hi; END;
    END FOR;

    RETURN max_till_now;

  ELSE

      min_till_now := hi;

      (FOR sy IN next_states(s))
        IF(min_till_now MIN := B(sy,lo,hi)) <= lo THEN RETURN lo; END;
      END FOR;

      RETURN min_till - now;

  END IF;

  END PROC B;
```

In the first loop of (4) the quantity max_till_now is never larger than hi or less than lo; hence we have

```
B(s,lo,hi)  MAX max_till_now = A_can_win(s) MIN hi MAX lo MAX max_till_now
                             = A_can_win(s) MIN hi MAX max_till_now
                             = B(s,max_till_now,hi).
```

Similarly, in the second loop of (4) we always have B(s,lo,hi) MIN min_till_now = B(s,lo,min_till_now). Moreover, all the recursive calls to B appearing in (4) occur in contexts in which B can as well by replaced by (B MIN hi MAX lo). Hence (v MIN lo MAX hi) can be replaced by v in the second line of (4). These remarks show that the following recursive procedure B2 satisfies B2(s,lo,hi) MIN lo MAX hi = B(s,lo,hi):

```
(5)     PROC B2(s,lo,hi);

        IF (v := A_wins(s)) /= OM THEN RETURN v; END;

        IF has_turn = 'A' THEN

          till_now := lo;

          (FOR sy IN next_states(s))

         IF (till_now MAX := B3(sy,till_now,hi)) > hi THEN RETURN hi; END;

        END FOR;

        ELSE

         till_now := hi;

         (FOR sy IN next_states(s))
           IF(till_now MIN := B2(sy,lo,till_now)) <= lo THEN RETURN lo; END;
         END FOR;

       END IF;

       RETURN till_now;

       END PROC B2;
```

The fact that the loops in (5) are terminated 'early', i.e terminated as soon as till_now rises to hi or sinks to lo, sometimes improves the efficiency of (3) very substantialy; this is what we want. Of course, we can exploit the symmetry of B2 to write it more compactly:

```
(6)     PROC B3(s,lo,hi); $ A polished 'alpha_beta' algorithm

        IF (v := A_wins(s)) /= OM THEN RETURN v; END;

        IF has_turn = 'B' THEN [hi,lo] := [-lo,-hi]; END;

        till_now := lo;
```

```
(FOR sy IN next_states(s))
  IF (till_now MAX := B3(sy,till_now,hi)) >= hi THEN RETURN hi; END;
END FOR;

RETURN IF has_turn = 'B' THEN -till_now ELSE till_now END;

END PROC B3(s,lo,hi);
```

IF LARGE designates any sufficiently large quantity, then; B3(s,-LARGE,LARGE) will be equal to A_can_win(s). It is convenient to represent such an 'infinitely large' quantity by OM, and also convenient to replace (6) by a recursive procedure yielding the value

```
IF has_turn = 'A' THEN B2(s,lo,hi) ELSE -B(s,-lo,-hi) END
```

Doing this gives us our next form of the alpha-beta procedure, namely

(7)
```
MACRO reverse(x); IF x = OM THEN OM ELSE -x END ENDM;

PROC A_can(s,lo,hi); $ second form of alpha_beta algorithm

till_now := lo;

(FOR sy IN next_states(s)) $ note that next_states(s) = { }
                           $ if s is a terminal state

  till_now := IF till_now = OM THEN a_can(sy,reverse(hi),reverse(lo))
     ELSE till_now MAX A_can(sy,reverse(hi),reverse(lo)) END;

  IF hi/=OM AND till_now >= hi THEN RETURN till_now; END;
 END FOR;

 RETURN IF(v := A_wins(s)) = OM THEN -till_now
      ELSEIF has_turn(s) = 'A' THEN v ELSE -v END;

 END PROC A_can; $ A_can_win(s) = A_can(s,OM,OM)
```

A close analysis sof algorithm (7) will show that it can be expected to derive the value of the A_can_win function for a tree of moves 2d levels deep in roughly the time that algorithm (3) would require to analyse a tree d levels deep. (Unfortunately, the necessary analysis is too complicated to be included in the present text). However, in spite of this very substantial improvement, complex games will still lead to trees of moves which are so deep and branch so rapidly that full exploration using algorithm (7) is quite impossible. One technique used to cope with this fundamental difficulty is to limit the number of recursive levels explored using (7). When this limit is reached, we use some ad-hoc estimate, called an evaluation heuristic, to approximate the value of A_can_win(s). In effect, this approach pretends to replaces the full game that we would like to analyse by a truncated game that is played for some limited number L of moves and then terminated with a payoff determined by the evaluation heuristic. To play the full game, we then reanalyse this truncated game each time it is a given player's turn to move and choose the best move in

the truncated game as his recommended move in the real game. Assuming that A_estimate(s) is the; estimated value of state s to player A, it is easy to modify (7) to incorporate such a limit on the number of levels of move and counter move that will be examined. Doing so, we get:

(8)
```
MACRO reverse(x); IF x = OM THEN OM ELSE -x END ENDM;

PROC Est_A_can_win(s,lo,hi,lim); $ alpha-beta algorithm with
                                 $ limited search

IF (lim -:= 1) = 0 THEN
 RETURN IF has_turn(s) = 'A' THEN A_estimate(s)
                        ELSE -A_estimate(s) END;
END IF;

till_now := lo;

(FOR sy IN next_states(s))

    till_now := IF till_now = OM THEN
    Est_A_can_win(sy,reverse(hi),reverse(lo),lim)
    ELSE till_now MAX Est_A_can_win(sy,reverse(hi),reverse(lo),
                                                    lim) END;
IF hi/=OM AND till_now >= hi THEN RETURN till_now; END;
END FOR;

RETURN IF (v := A_wins(s)) = OM THEN -till_now
       ELSEIF has_turn(s) = 'A' THEN v ELSE -v END;

END PROC Est_A_can_win;
```

(Application of the above to a simple game, e.g. KALAH, should be inserted here).

11.9 Implementation of a Macroprocessor

In this section we will show how to implement the SETL macro feature described in Section 6.4. The context within which this macroprocessor is to be implemented is assumed to be as follows:

(i) The macroprocessor reads a succession of tokens, obtained by decomposing some input file into successive tokens.

(ii) When the special token MACRO is encountered, a macro definition is opened. This token must be followed by a macro-name, which can in turn be followed by a list of formal parameters and generated formal parameters, in the manner explained in Section 6.4.4, 6.4.5. The macro-body following such a 'macro opener' is collected, and saved in a map -def_of-, which associates each macro name with its list of parameters, its list of generated parameters, and its macro body.

(iii) When a macro invocation starting with a token belonging to the domain of the map -def_of- is encountered, its actual arguments are collected, and the invocation is replaced by a substituted version of the macro body. This

substituted text is logically inserted immediately in front of the remainder of the input file, and reprocessed by the macro-expansion mechanism, thereby ensuring that macro invocations and definitions embedded within macro bodies will be treated in the manner described in Section 6.4.6.

(iv) The macroprocesor makes various syntactic checks. For example, it checks that the parameters appearing in a macro definition are all distinct, and that each macro invocation has as many arguments as the corresponding macro-definition has parameters. If an error is detected, a diagnostic message is printed, and any macro-action in progress is simply bypassed.

(v) The macroprocessor is structured as a MODULE, which exports just one procedure, namely a parameterless procedure called -next_tok-, which can be called repeatedly to obtain the sequence of tokens representing the input file after macro- expansion. When the input file is exhausted, -next_tok- will return OM. The macroprocessor MODULE imports just one procedure, namely a parameterless procedure called -input_tok-. Successive calls to input_tok generate the sequence of input tokens which constitute the macroprocessor's initial input.

```
    MODULE language_processor - macroprocessor;

    EXPORTS next_tok;
    IMPORTS input_tok;

    VAR
     def_of,              $ maps macro-names into their definitions
     expanded_toks;       $ vector of tokens obtained by prior
macro-expansion
     INIT
        gmac_ctr := 0      $ generated macro argument counter
        def_of := 0        $ initially no definitions
        expanded_toks := []; $ initially no prior tokens

    CONST Illformed_list = 'ILLFORMED MACRO PARAMETER LIST';;
                           $ error message

    PROC next_tok;         $ called to obtain successive tokens in the
                           $ sequence of tokens generated by macro expansion

LOOP DO       $ we return to this point whenever
                    $ macro-errors are detected

    IF (tok := another_tok( )) = OM THEN RETURN OM; END;
                         $ end of input file encountered

    IF (tok /= 'MACRO') AND (mdef := def_of(tok)) = OM THEN
        RETURN tok;           $ token is ordinary;
    END IF;

    IF tok = 'MACRO' THEN  $ start new macro definition

        IF (parm_list := get_parm_list( )) = OM
           OR (mac_body := get_macro_body( )) = OM THEN
```

```
            GOTO try_again;    $ since macro is bad

        END IF;

        [mac_name,mac_pars,mac_gpars] := parm_list;
               $ get macro name and parameters

        def_of(mac_name) := (IF mac_body = [ ] THEN OM   $ macro drop
          ELSE [mac_pars,#mac_gpars,template(mac_body,mac_pars,mac_gpars)]
END;

    ELSE                    $ macro invocation

        [mac_pars,n_gpars,mac_template] := mdef;
               $ look up macro-definition

        IF (arg_list := get_arg_list(#mac_pars)) = OM THEN
            CONTINUE                    $ abort expansion
               $ since number of arguments and number of parameters differ
        END IF;

        (FOR n IN [1..n_gpars]) mac_pars WITH := [generated_parm( )]; END;
           $ generate additional parameters as required

           $ next replace the macro at the start of the expanded_tokens
           $ vector by its expansion

        expanded_tok :=
        +/[IF is_string (mac_tok) THEN [mac_tok] ELSE mac_pars(mac_tok) END:
                  mac_tok = mac_template(j)] + expanded_tokens;

     END IF tok;

     END LOOP;          $ now that macro has been expanded, try again to
                        $ supply the requested token

     END PROC next_tok;

    PROC another_tok; $ 'token feeder' for macro-processor

        $ This returns the token standing at the head of
        $ -expanded_toks- unless expanded_toks is empty,
        $ in which case it calls the 'primary'
        $ token source -input_tok- to get the token to be returned.

    RETURN IF (tok FROMB another_tok)/= OM THEN tok
            ELSE input_tok( ) END;

    END PROC another_tok;

    PROC get_parm_list; $ gets sequence of parameters for macro

       $ The sequence of parameters collected by this procedure must
       $ be a comma_separated list opened by a left parenthesis and
       $ closed by a right parenthesis.  If this syntax is violated,
```

```
      $ or if two parameters are identical,
      $ an error message is printed, and OM is returned.

   have_parms := FALSE; $ flag: No generated parameters yet

   mac_parms := mac_gparms := [ ]; $ initializes parameters and
                                   $ generalized parameters

   IF (tok := another_tok( )) = ';' THEN RETURN parms; END;
                        $ no parameters

   MACRO check(condition,msg);
       IF NOT (condition) THEN RETURN err_msg(msg); END;
   ENDM;

   check(tok = '(', Illformed_list);

   (UNTIL tok = ')')     $ until terminating parenthesis

       check((tok := another_tok( )) /= OM, illformed_list);
       mac_parms WITH := tok;
       check((tok := another_tok( )) = ',' OR tok = ')', illformed_list);

   END UNTIL;

   RETURN [name,mac_parms,mac_gparms];

   END PROC get_parm_list;

   PROC err_msg(message); $ error message routine

   print(message); $ print error message

   RETURN OM;       $ signal error

   END PROC get_parm_list;

   PROC get_macro_body; $ collects sequence of tokens to ENDM

   body := [ ];

   (WHILE (tok := another_tok( )) /= 'ENDM');
       check(tok /=  OM, 'MACRO BODY NOT PROPERLY ENDED');
       body WITH := tok;
   END WHILE;

   RETURN body;

   END PROC get_macro_body;

   PROC template(mac_body,mac_pars,mac_gpars);

   $ This procedure builds the 'macro template' stored as the
   $ definition of a macro.  The template consists of the
   $ string constituting the macro body, but with every
```

```
   $ parameter and generated parameter replaced by an integer.

   counter := 0; $ start count at zero

   replacement := {[t,(counter +:= 1)]: t IN mac_pars + mac_gpars};
      $ This maps every macro parameter into its replacement

   RETURN [replacement(t)?t: t IN template];

   END PROC template;

   PROC generated_parm; $ auxiliary procedure-produces generated
                        $ macro parameters.

   $ The macro parameters generated by this procedure have
   $ the form 'ZZZn', where n is the string representation;
   $ of an integer.

   RETURN 'ZZZ' + STR(gmac_ctr +:= 1);

   END PROC generated_parm;

END MODULE;
```

11.11 Exercises

Ex. 1 A 'nondeterministic' Turing machine is a Turing machine TM whose action mapping is not constrained to be single-valued. In addition, one particular internal state of each such machine must be designated as its 'failed' state. Such machines can be regarded as describing indefinitely large families of computations which proceed in parallel. More specifically, we start with a given tape, tape position, and internal machine state, as in the case of an ordinary Turing machine. Then, whenever the internal -state- and the -character- under the machine's read head are such that action(character, state) is multivalued (consisting, say, of n values), we create as many logical copies of the machine as needed, and assign one of them to take each of these n actions and continue the computation. This can generate a rapidly expanding set of computations, all proceeding in parallel. If a particular logical copy TMj of TM reaches the special 'failed' internal state, the particular path of computation which it is following ceases, and TMj is simply deleted. As soon as any computation TMk reaches an ordinary 'stop' condition all other computations are deleted, and the result calculated by this 'successful' logical copy TMk of TM becomes the final result of the nondeterministic computation. On the other hand, if all computations TMk reach the 'failed' internal state, the nondeterministic Turing machine computation is considered to have failed.

Modify the Turing machine simulation program shown in Section 11.6 so that it can simulate both ordinary and nondeterministic Turing machines.

Ex. 2 A 'multi-tape' Turing machine is one which has several separate tapes, a read-write head on each, and whose action on each cycle determined by its internal state and by the characters found under all of its

read-write heads. Modify the Turing machine simulation program shown in Section 11.6 so that it can simmulate multitape Turing machines with any specified number of heads.

Ex. 3 Can you think of any well-defined computing automaton or computational process whose activity could not be simulated by a SETL program? Review Exercises

1 and 2 before you answer.

Ex. 4 The macroprocessor shown in Section 11.9 is programmed to imitate the present SETL macroprocessor, which regards every comma in a macro argument list as a separator. For example, if -my_mac- is a macro name, then the invocation

```
my_mac(f(x,y),z)
```

is considered to have three components, namely

```
f(x        y)     z
```

This is not the best convention: it would be better to regard commas contained within parentheses or brackets as being invisible to the macroprocessor, so that the macro-call shown above would be regarded as having just two arguments f(x,y) and z. Modify the macroprocessor so that it behaves in this way.

Ex. 5 (Continuation of Ex. 4). Especially if the modification suggested in Exercise 4 is made, use of a macroprocessor becomes subject to two dangers:

(a) If the parenthesis terminating an argument list is missing, much of the body of text following a macro invocation may be swallowed up in what appears to be a very long final argument.

(b) If the keyword 'ENDM' ending a macro is missing or mispelled, the text following a macro definition may appear to be swallowed up by the macro-definition.

Modify the macro-processor of Exercise 4 so as to limit each macro argument to 50 tokens and each macro-definition to 200 tokens.

Exercises related to the 'check processing' system of Section 5.4.3

Ex. 6 Modify the check processing system so that it tracks

(a) the total dollar volume of transactions handled each day.

(b) The total dollar credit/debit that the bank using the system has built up against each of its correspondent banks.

These quatities should be printed out as additional information by the DAY transaction.

Ex. 7 Modify the check processing system, adding a new transaction DEL which prints out a list of all accounts for which more than a month has gone by without at least 10% of a customer's outstanding overdraft_debit having been paid.

Ex. 8 Modify the check processing system, adding the following two transactions:

(a) A transaction AB ('abuse') which shows all accounts for which an excess overdraft has accumulated or against which more than 10 'insufficient funds' charges have been made during the current month.

(b) A transaction I ('idle') which shows all accounts against which no checks have been drawn during the past six months.

Ex. 9 Modify the check processing system, adding transactions O and CL which allow new customer accounts to be opened and closed. Closing of accounts should be handled carefully: such accounts should be marked as having been closed, but should not actually be deleted while there exist outstanding transactions, still to be returned by other banks, that might affect the account which is being closed.

When an account is finally closed, the balance remaining in it should be used to pay off any outstanding overdraft_debit, and a check for the amount remaining in the account after this final payment should be prepared for mailing. How will you handle an account closing when the balance remaining is insufficient to pay off the overdraft-debit?

Ex. 10 Modify the check processing system so that it can add a short advertisement to the monthly statements being prepared for mailing to customers. The text of this advertisment should be supplied by a transaction of the form.

ADVERT n

where n is an integer, and where this line will be followed by n more lines giving the text of the advertisment. This transaction must be run just before the DAY transaction which triggers preparation of monthly statements.

Ex. 11 If you have a checking account, save the next monthly statement you get from your bank, and scrutinize it carefully. How may of the features of this statement suggest that your bank is using a program similar to the check processing program shown in Section 5.4.3? What features reveal the use of processing steps that our simplified check processing system does not perform? If you can find any such feature, choose one of them and mofify the check processing system to include it.

Ex. 12 Modify the check processing system so as to make it a model for the activity of several banks. Each of these banks will run the modified check processing system once per day, generating files of messages which are then sent to the other banks in the system and added to the transaction files that these banks will process during their next day's run. Execute your

modified program with appropriate inputs so as to simulate several day's activity for the whole 'financial system'.

Ex. 13 The degree of compression attained by the Huffman coding procedure shown in Section 11.7 can be increased by using the fact that the probability of encountering a character depends on the character that has just been encountered. That is, we can calculate not one, but a whole family of Huffman trees, one for each high-probability character c in our alphabet; this tree should position other characters d according to probability that d follows c.

Develop a modified Huffman package which uses these more refined probabilities, and also a modified -Huff_tree- code which calculates all the Huffman trees required.

Ex. 14 If the 'Huff' and 'Puff' procedures shown in Section 11.7 are really to be used for compressing large texts, we will want them to produce densely packed character strings rather than SETL-level sequences of zeroes and ones. To achieve this without having to abandon SETL in favor of a language in which sequences of bits can be manipulated directly, we can break the sequence of zeroes and ones that 'Huff' would most naturally produce into eight-bit sections, each of which is then represented by a single SETL character. Conversely, when decoding, we can first convert each character in the string being decoded into a string of zeroes and ones.

Modify the Huffman routines shown in Section 11.7 so that they work in this way. Your modified -setup- procedure should construct the extra data structures needed to convert characters into 8-bit sequences of zeroes and ones, and vice-versa.

Ex. 15 (Continuation of Exercise 14) The decoding procedure shown in Section 11.7 and further described in Exercise 14 can be accelerated keeping a map -Decode- which sends the start (say the first eight bits) of the sequence s being decoded either into a pair [c,n], where c is the first character obtained by decoding s and n is the number of bits of s that represent this character, or into the -node- of the Huffman tree that is reached reach after walking down the tree in the manner determined by the first 8 bits of s, if these 8 bits do not lead us to a terminal node. Rewrite these routines by incorporating the suggested improvements.

Ex. 16 The Huffman -setup- procedure shown in Section 11.7 can be made more efficient by saving the sequence of zeroes and ones describing the path from each Huffman tree -node- traversed. This information can be stored at the node. This makes it unnecessary for the -setup- procedure to traverse any edge of the Huffman tree more than once. Rewrite -setup-, incorporating this improvement.

Ex. 17 The Huff_tree procedure shown in Section 11.7 can be made more efficient by using the tree-like structures described in Section 11.7 to accelerate the auxiliary -get_min- procedure. Rewrite -Huff_tree- and -get_min-, incorporating this improvement.

Ex. 18 (Continuation of Exercise 13). Storing a Huffman tree requires memory space proportional to the size of the alphabet whose characters are attached to the terminal nodes of the tree. If the improved technique

described in Exercise 13 is used, such a tree will have to be stored for each character in the alphabet, and the amount of space required for this can grow unpleasantly large (especially if the data compression procedure is to be reprogrammed for a small machine). In this case, the following expedient can be used to reduce the amount of storage required:

(a) For each character c, establish a limit L(c) which will bound the number of nodes used in the modified Huffman tree built from the frequency count developed for letters following c. This limit should be larger for commonly occuring characters c, smaller for infrequent characters.

(b) For each c, find the L(c) characters which most frequently follows c, and 'lump' all the other characters into a new character c'. The sum of the frequencies of all these 'lumped' characters then becomes the frequency of c'.

(c) Build a Huffman tree for the alphabet of L(c) + 1 characters left after step (b). Then let the code of each character not 'lumped' into c' be determined as in Exercise 13, but let the code of each character x 'lumped' into c' be the concatenation of the normal Huffman code of c' with the standard internal SETL code of c.

Modify the Huffman encode/decode procedures developed in Exercises 13, 14, and 15 to incorporate this space-saving refinement.

Ex. 19 Develop the auxiliary procedure pref_sort(t,pref) invoked by the -assign- procedure of Section 11.3. This should sort a tuple t into the order defined by a Boolean-valued function pref(s1,s2) which returns the value TRUE if s1 should come before s2 , FALSE otherwise. Your sorting routine should be modeled after either 'mergesort' or 'quicksort'.

Ex. 20 In playing a game, one may wish not only to win as much as possible, but also to win in the smallest possible number of moves. A recursion much like formula (1) of Section 11.8 can be used to determine the minimum number of steps which the winning player will need to bring the game to a successful conclusion. Find this recursion, and use it to develop a variant of the 'alpha-beta' game-playing procedure which tells the winning player how to win as rapidly as possible, and tells the losing player how to postpone his inevitable defeat as long as possible.

Ex. 21 The 'alpha-beta' game playing program (see Est_A_can_win, Section 11.8) operates most efficiently if moves likely to return a large Est_A_can_win value are explored first. To guess in advance which moves these are likely to be, once can save the values calculated by Est_A_can_win during each cycle of play, and use these values as estimates of move quality the next time it is the same player's turn to move. Write a variant of the Est_A_can_win procedure which incorporates this improvement.

$

HAPTER 12

INDEX

$

APPENDIX A

SETL RESERVED WORDS

The following words have a predefined meaning within a SETL program, and should only be used for their defined purpose.

ABS	FIX	MIN	REPR
ACOS	FLOAT	MMAP	RETURN
ALL	FLOOR	MOD	REWIND
AND	FOR	MODE	RMATCH
ANY	FORALL	MODULE	RNOTANY
ARB	FROM	NARGS	RPAD
ASIN	FROMB	NEWAT	RSPAN
ASSERT	FROME	NOT	RW
ATAN	GENERAL	NOTANY	SET
ATAN2	GET	NOTEXISTS	SETEM
ATOM	GETB	NOTIN	SIGN
BACK	GETEM	NOTRACE	SIN
BASE	GETF	NPOW	SMAP
BOOLEAN	GETIPP	ODD	SPAN
BREAK	GETK	OF	SPARSE
CALLS	GETSPP	OK	SPEC
CASE	GOTO	OM	SQRT
CEIL	HOST	OP	ST
CHAR	IF	OPEN	STATEMENTS
CLOSE	IMPL	OPERATOR	STEP
CONST	IMPORTS	OR	STOP
CONTINUE	IN	PACKED	STR
COS	INCS	PASS	STRING
DATE	INIT	PLEX	SUBSET
DEBUG	INTEGER	POW	SUCCEED
DIRECTORY	IS_ATOM	PRINT	TAN
DIV	IS_BOOLEAN	PRINTA	TANH
DO	IS_INTEGER	PROC	TERM
DOING	IS_MAP	PROCEDURE	THEN
DOMAIN	IS_REAL	PROG	TIME
DROP	IS_SET	PROGRAM	TITLE
EJECT	IS_STRING	PUT	TRACE
ELMT	IS_TUPLE	PUTB	TRUE
ELSE	LEN	PUTF	TUPLE
ELSEIF	LESS	PUTK	TYPE
END	LESSF	QUIT	UNSPEC
ENDM	LEV	RANDOM	UNTIL

EOF	LIB	RANGE	UNTYPED
ERROR	LIBRARIES	RANY	VAL
EVEN	LIBRARY	RBREAK	VAR
EXISTS	LOCAL	RD	WHERE
EXIT	LOOP	READ	WHILE
EXPORTS	LPAD	READA	WITH
EXPR	MACRO	READS	WR
FAIL	MATCH	REAL	WRITES
FALSE	MAX	REMOTE	YIELD

$

Appendix B : syntax diagrams.

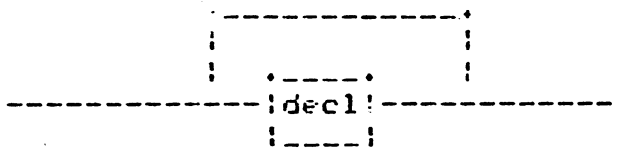
Throughout this text, syntax diagrams are used to describe the grammatical structure of SETL constructs. For convenience, all syntax diagrams appearing in the text are collected in this appendix.

Each diagram describes the structure of a language construct. Each path through a given diagram traces one valid instance of the corresponding construct. The following conventions are used in drawing a syntax diagram :

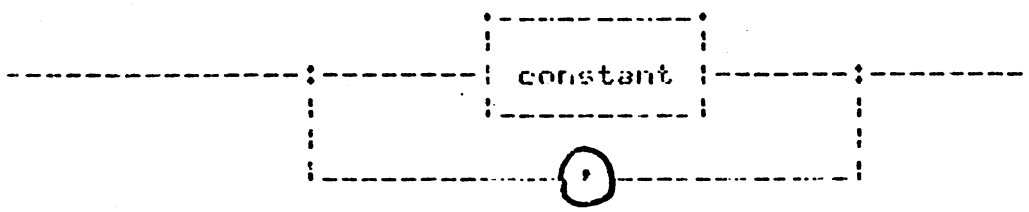
a) Syntactic classes a written in lower case and enclosed in rectangular boxes.

b) Terminal symbols of the language (delimiters and keywords) are capitalized, and enclosed in rounded boxes.

c) When the presence of a construct in a given diagram is optional (say the declarations in a program) then a path that bypasses the optional construct appears in the graph above that construct. For example, a procedure body includes the following :



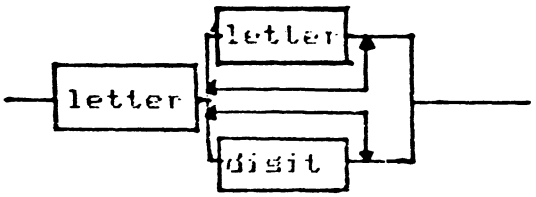
d) Repetition is indicated by a backwards path that passes under the repeated construct. For example, a list of constants is a sequence of one or more constants, separated by commas. The corresponding syntax graph for the construct 'constant list' is the following :



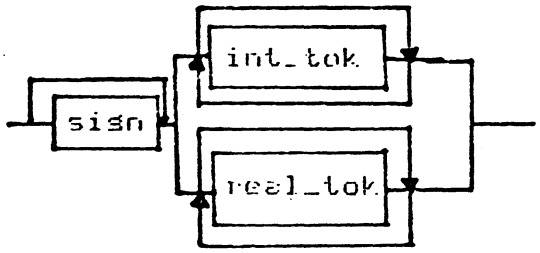
e) The end of composite statements (loops, IF- and CASE-statements) is indicated by the token END, optionally followed by one or more of the tokens that start te statement. The ellipsis (...) is used in the syntax diagrams to indicate the presence of such optional tokens.

The following graphs describe the structure of the valid tokens of the language.

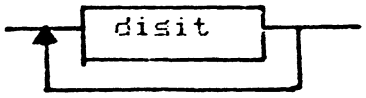
name



number

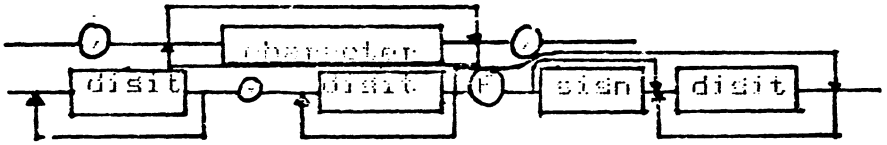


int_tok

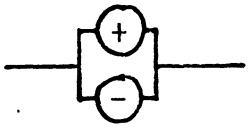


string

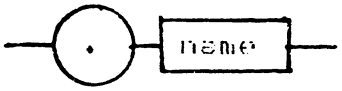
real_tok



sign



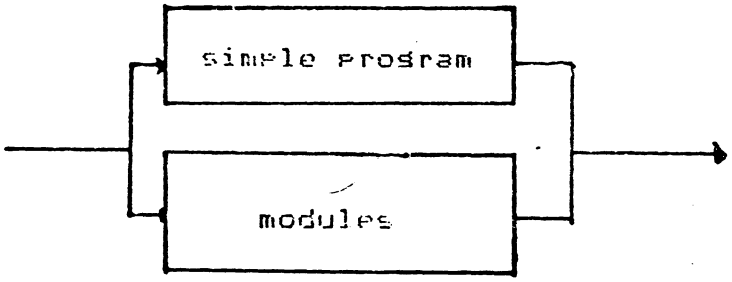
dot_tok



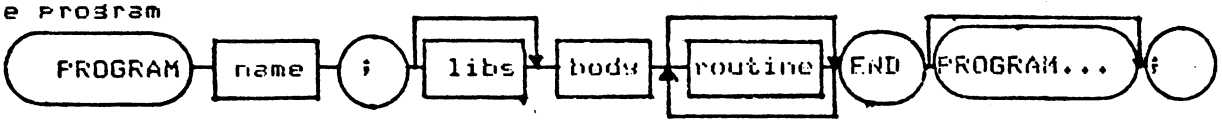
A.2 Program structure.

A SETL program is an instance of the construct -program-, together with any referenced libraries, which are instances of -lib-unit- .

program

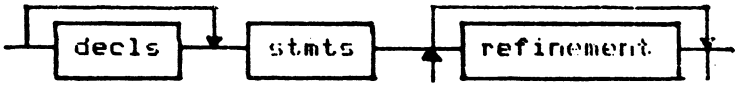


simple program

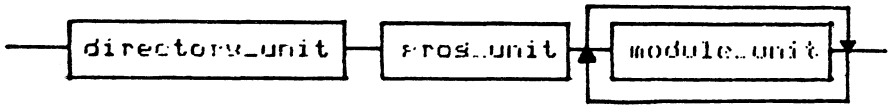


libs

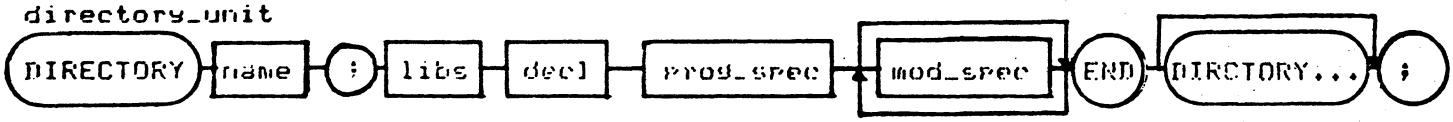
body



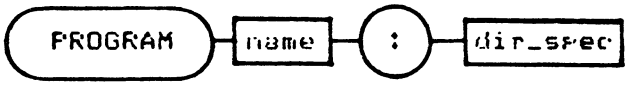
modules



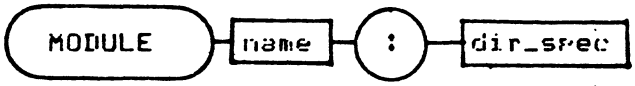
directory_unit



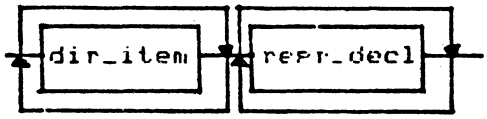
prog_spec



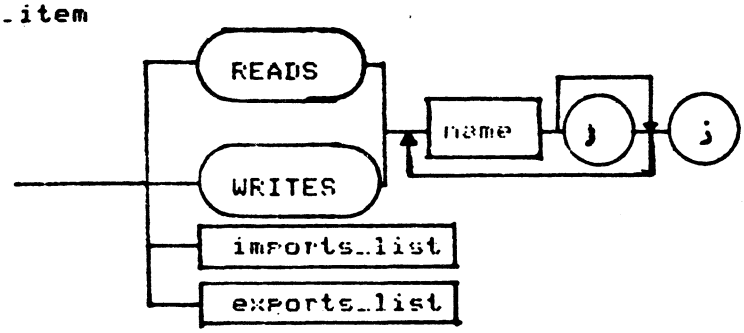
mod_spec



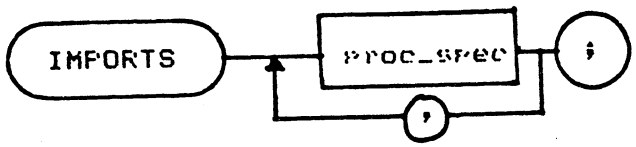
dir_spec



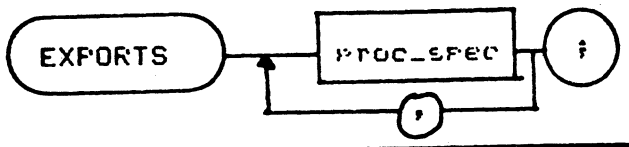
dir_item



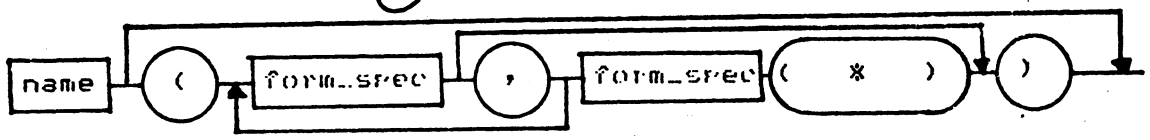
imports_list



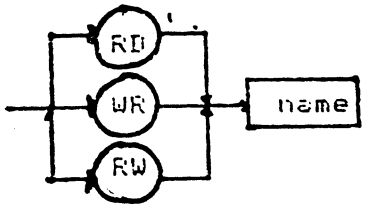
exports_list



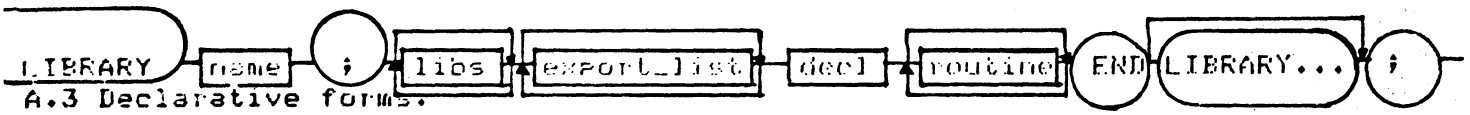
proc_spec



form_spec

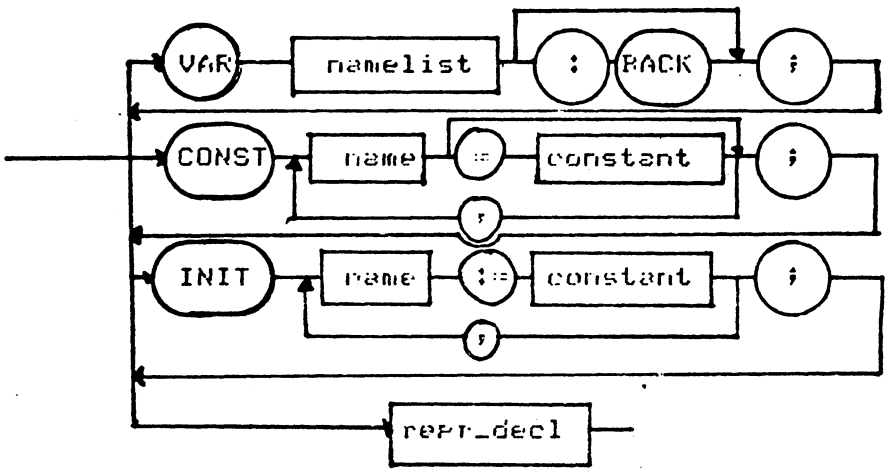


lib_unit

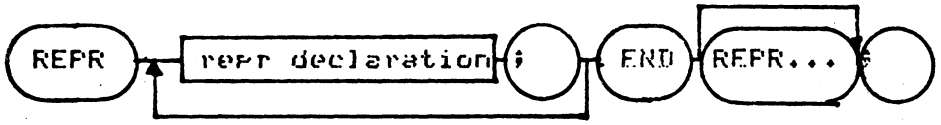


A.3 Declarative forms.

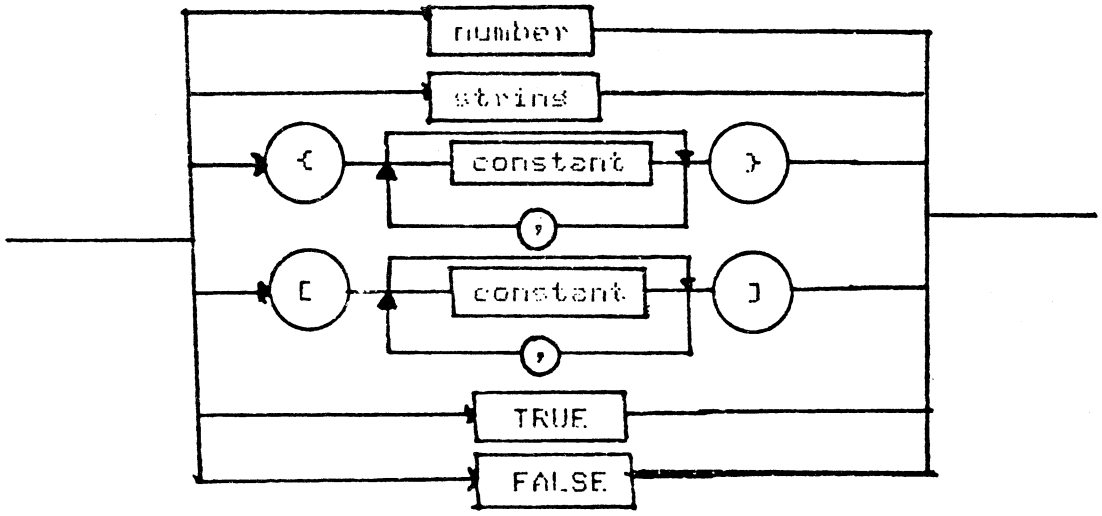
decls



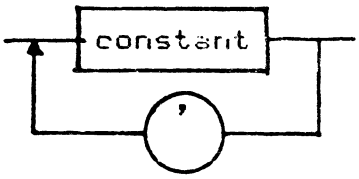
repr_decl



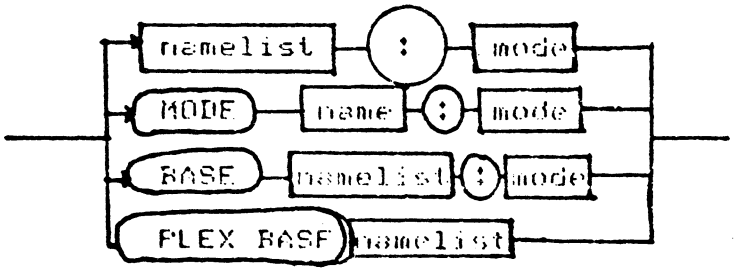
constant



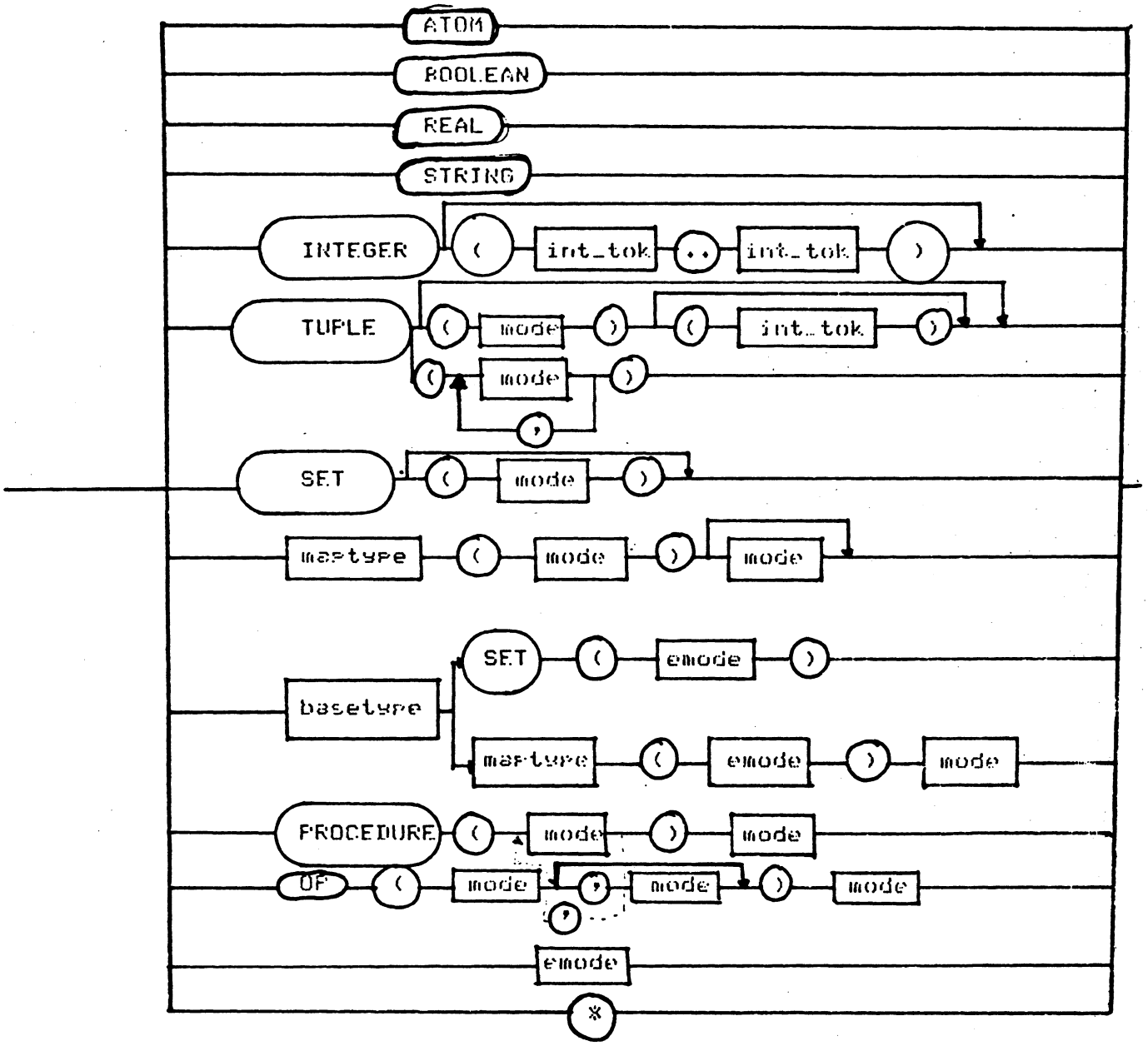
const_list



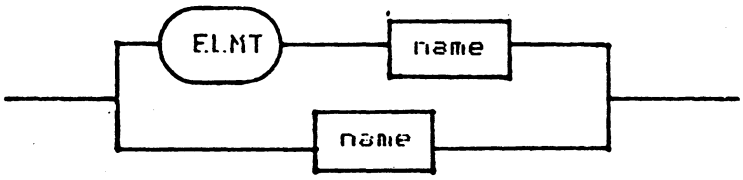
repr declaration



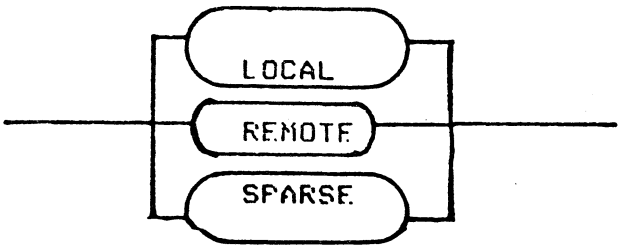
mode


ATOM
BOOLEAN
REAL
STRING
INTEGER
(
int_tok
..
int_tok
)
TUPLE
(
mode
)
(
int_tok
)
(
mode
)
,
SET
(
mode
)
maptype
(
mode
)
mode
SET
(
emode
)
basetype
maptype
(
emode
)
mode
PROCEDURE
(
mode
)
mode
OF
(
mode
,
mode
)
mode
,
emode
*

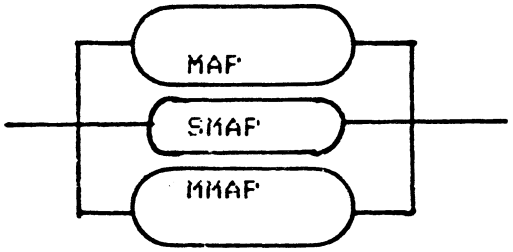
emode


ELMT
name
name

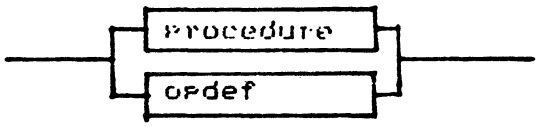
basetype



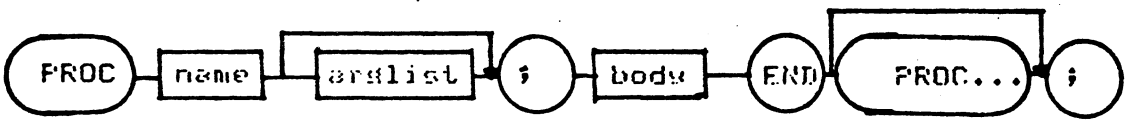
maptype



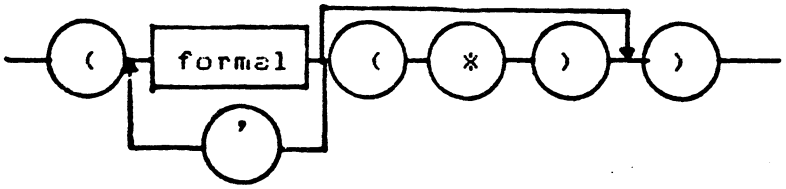
routine



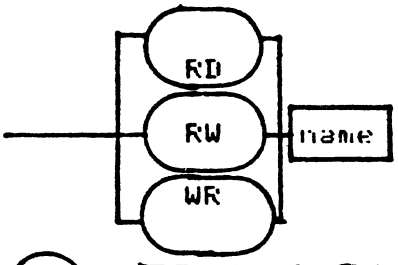
procedure



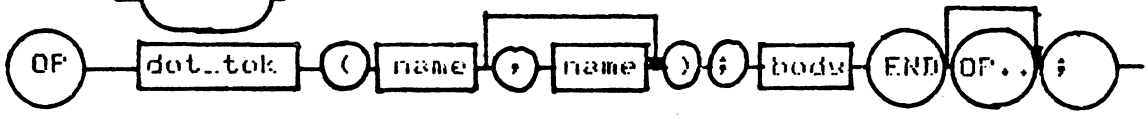
arglist



formal



opdef



refinement

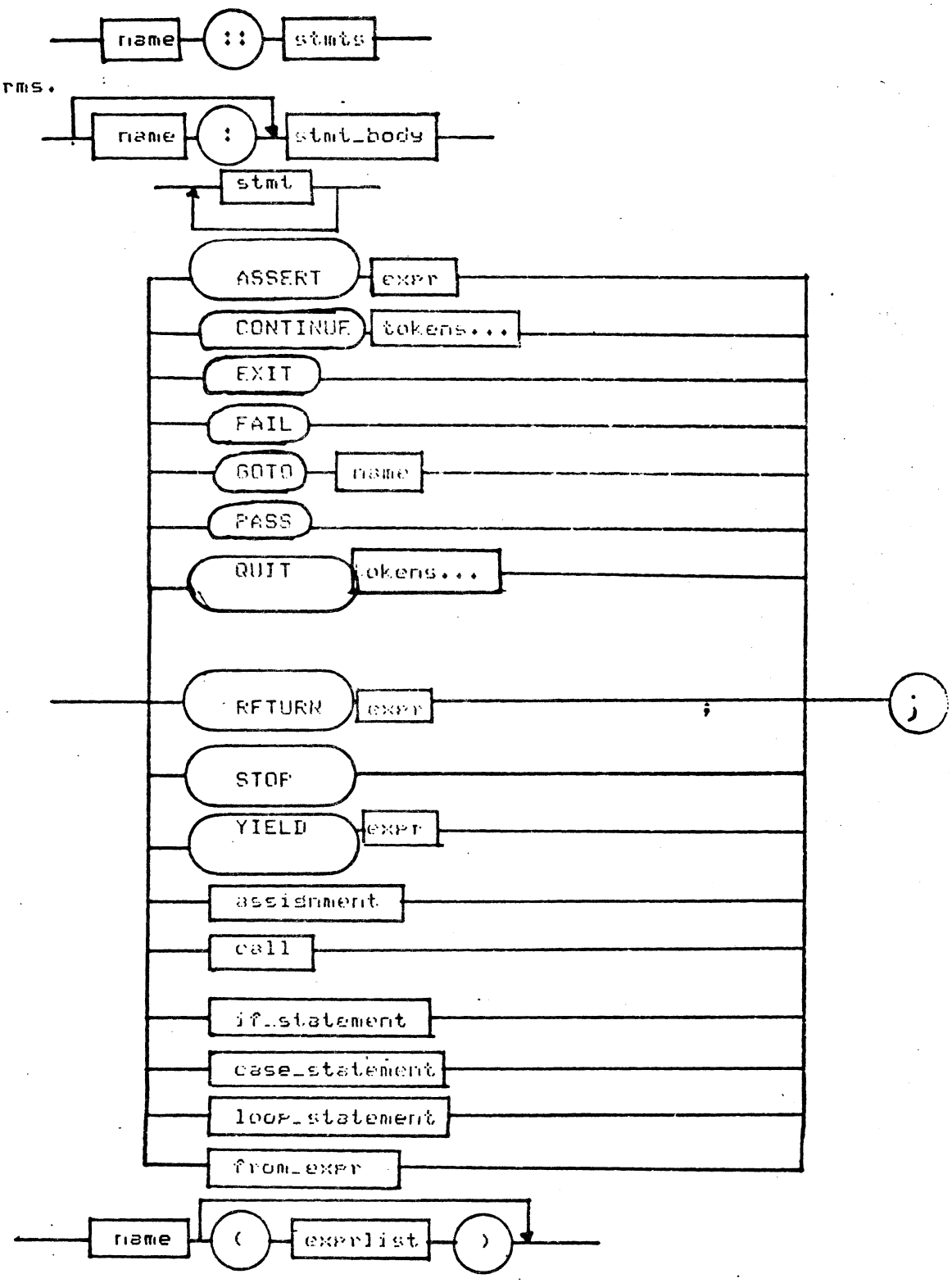
A.4 Statement forms.

stmt

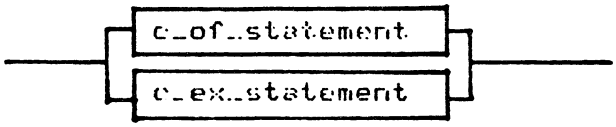
stmts

stmt_body

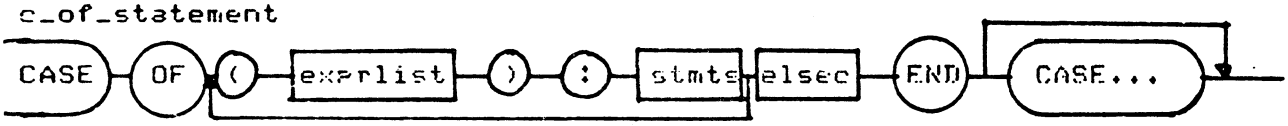
call



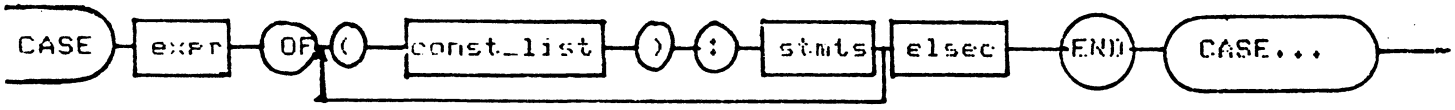
case_statement



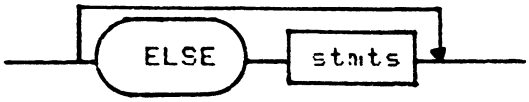
c_of_statement



c_ex_statement

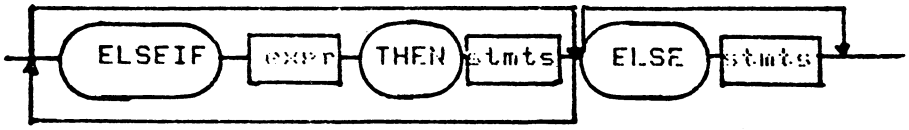


elsec

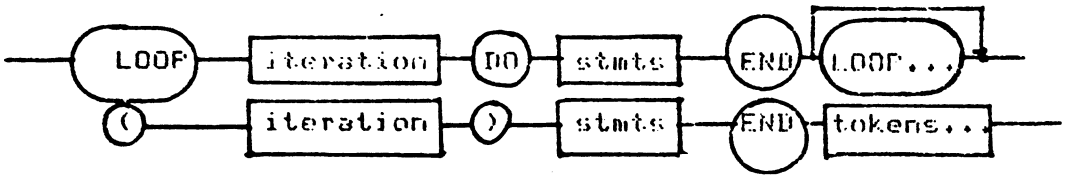


if_statement

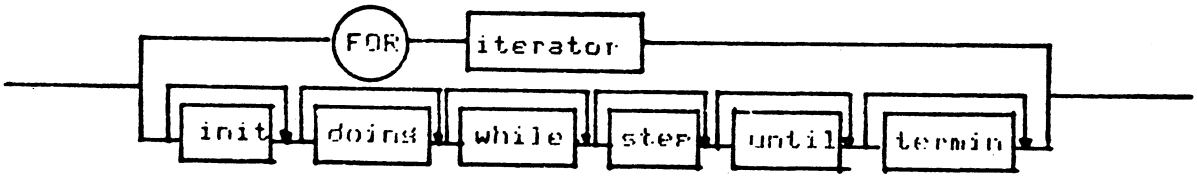
elseif



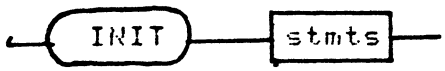
loop_statement



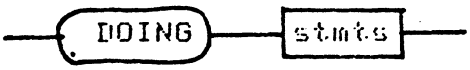
iteration



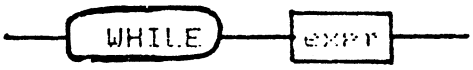
init



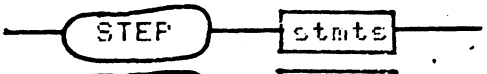
doing



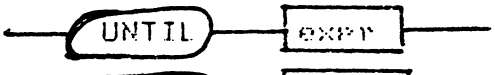
while



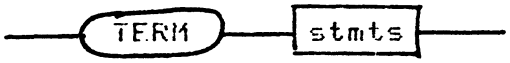
step



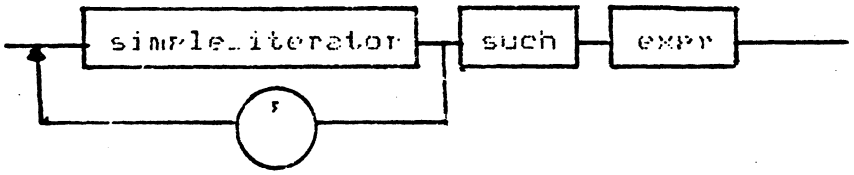
until



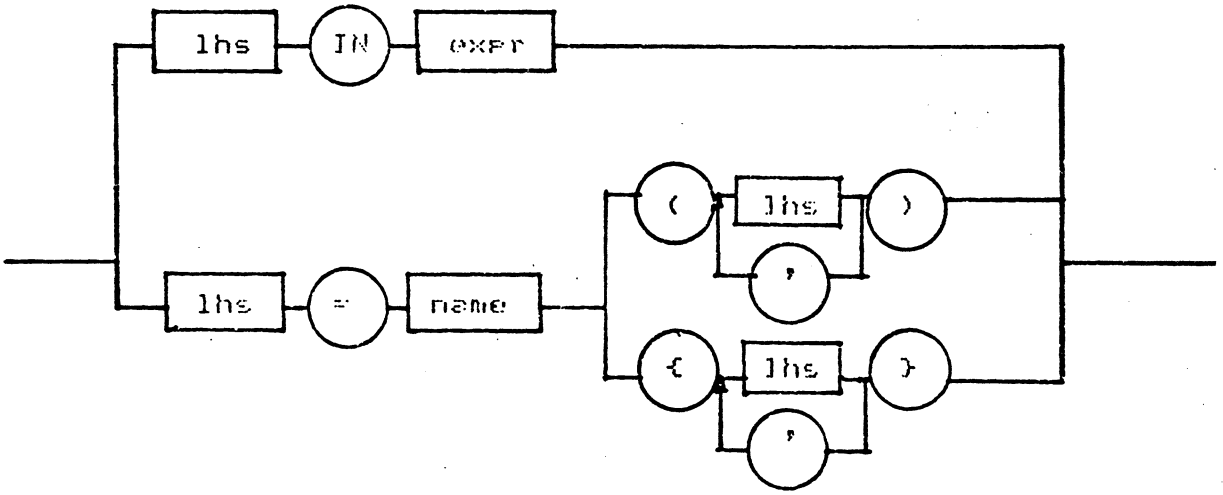
termin



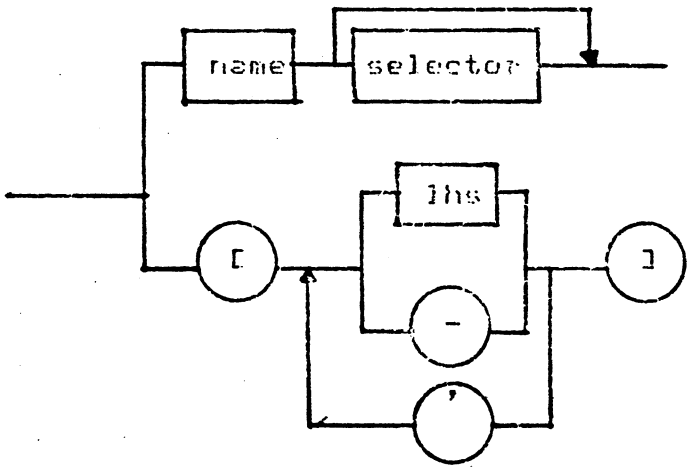
iterator



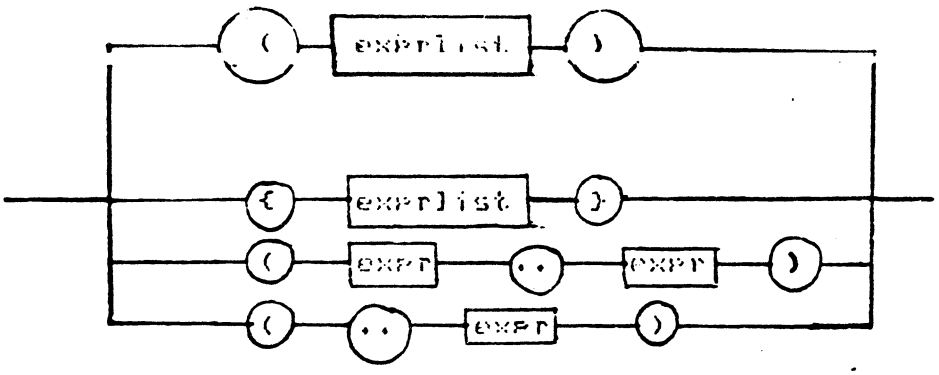
simple_iterator



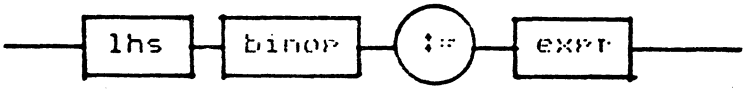
lhs



selector



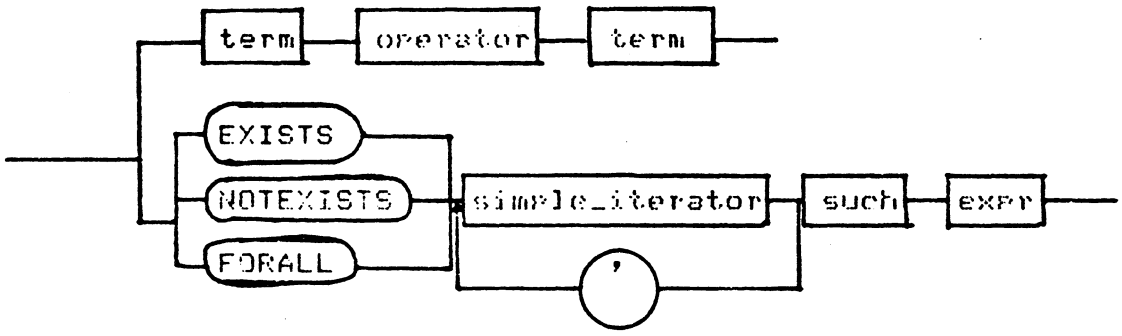
assignment_statement



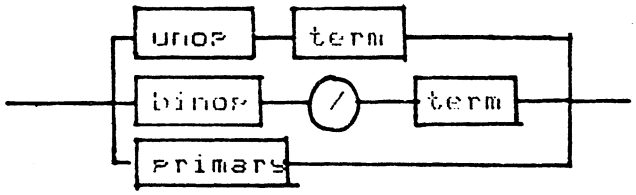
A.5 Expressions.

The following syntax graphs do not describe fully the relative precedence of operators. A complete table of operator precedences is to be found in Sec.3.xxx. The construct -binop- includes the predefined binary operators and the user-defined operators. Similarly -unop- refers both to predefined and user-defined operators.

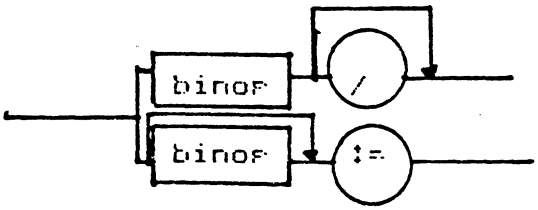
expr



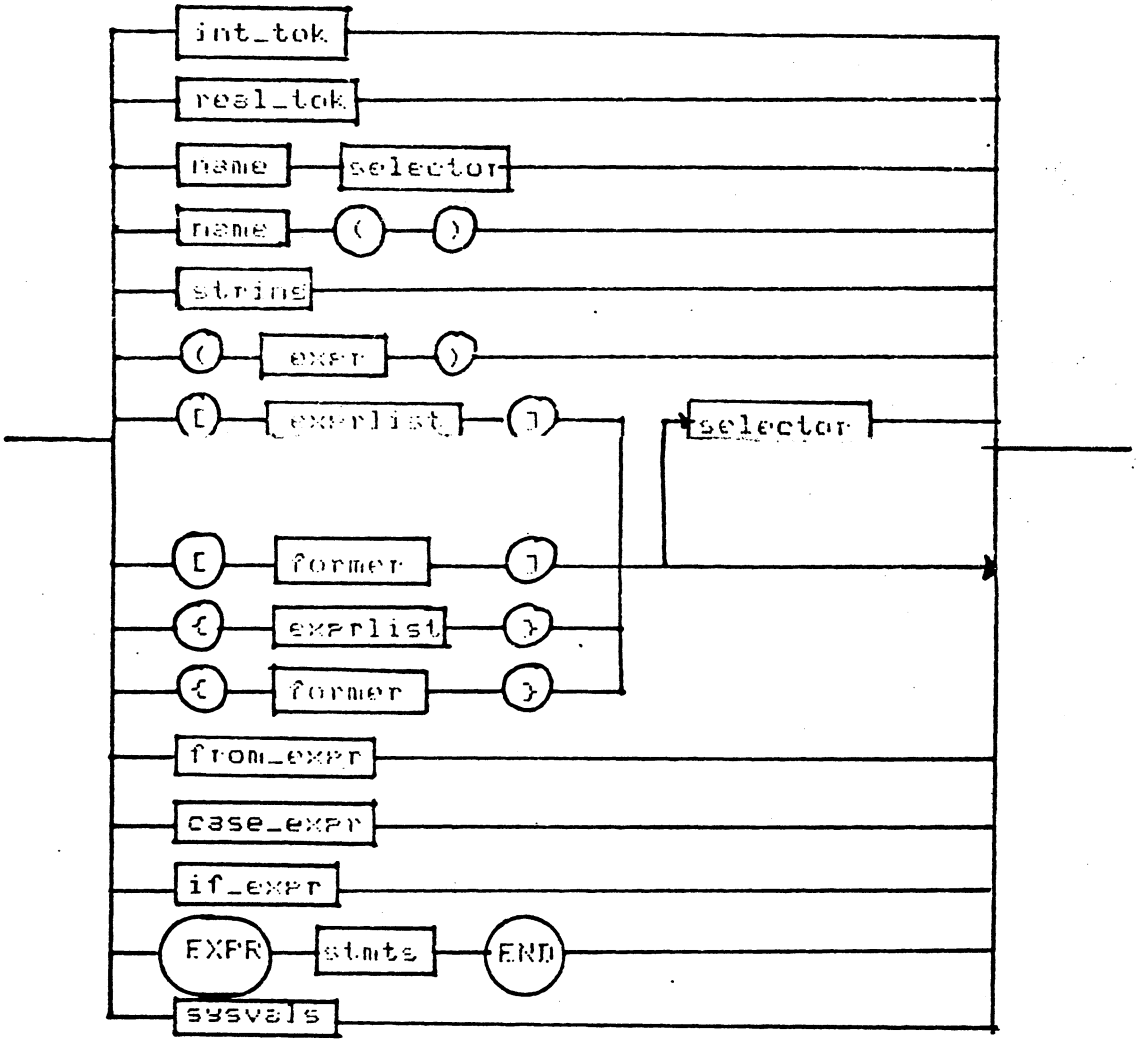
term



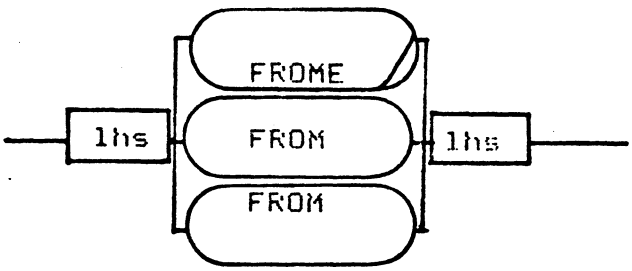
operator



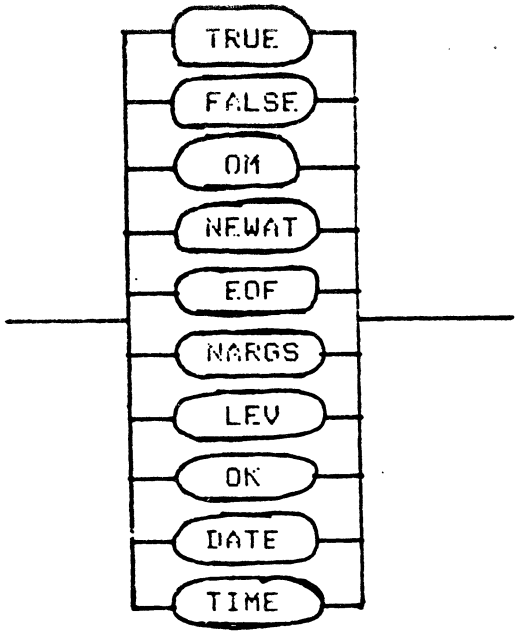
primary



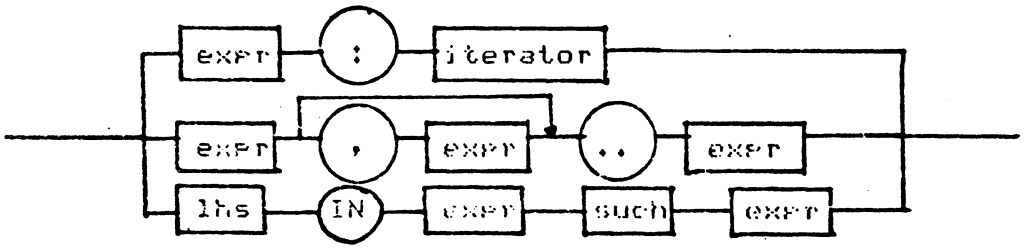
from_expr



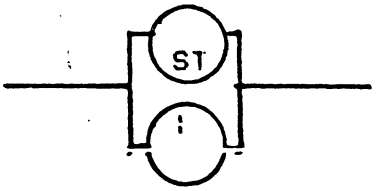
sysvals



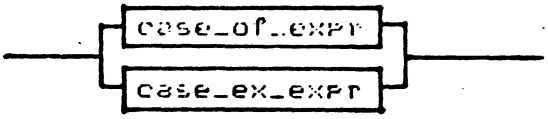
former



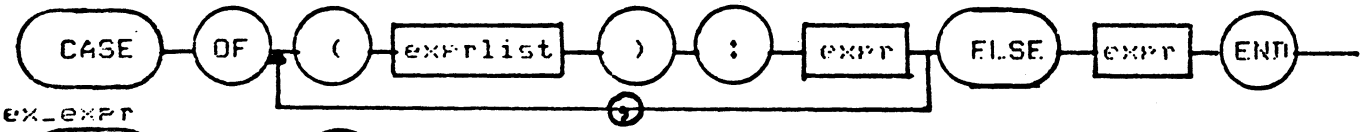
such



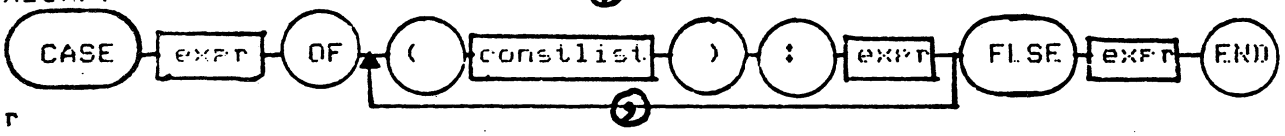
case_expr



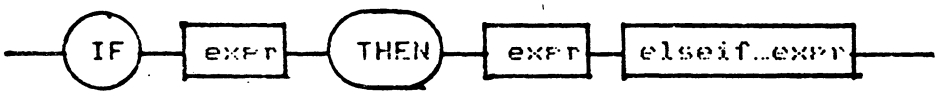
case_of_expr



case_ex_expr



if_expr



elseif_expr