ON PROGRAMMING

An Interim Report on the SETL Project

Part I: Generalities

Part II: The SETL Language and Examples of Its Use

(Parts I and II are consolidated in this volume)

Jacob T. Schwartz

Revised June 1975

Computer Science Department

Courant Institute of Mathematical Sciences

New York University

DISCLAIMER

This work was supported by the National Science Foundation Contract NSF-DCR 75-09218 and by the U. S. Energy Research and Development Administration, Contract E(11-1)-3077.

The Courant Institute publishes a number of sets of lecture notes. A list of titles currently available will be sent upon request.

Courant Institute of Mathematical Sciences
251 Mercer Street, New York, New York 10012

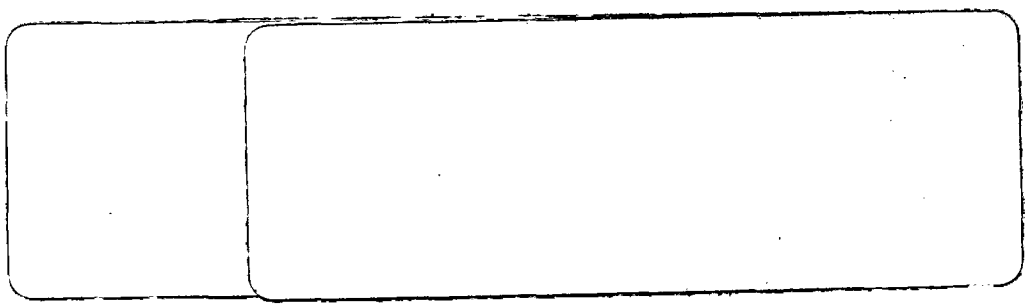


Table of Contents

THIS PAGE
WAS INTENTIONALLY
LEFT BLANK

Preface

The work of which the present manuscript gives first results has its roots in certain musings concerning the relationship between mathematics and programming in which the author has from time to time indulged. On the one hand, programming is concerned with the specification of algorithmic processes in a form ultimately machinable. On the other, mathematics describes some of these same processes, or in some cases merely their results, almost always in a much more succinct form, yet in a form whose precision all will admit. Comparing the two, one gets a very strong even if initially confused impression that programming is somehow more difficult than it should be. Why is this? That is, why must there be so large a gap between a logically precise specification of an object to be constructed and a programming language account of a method for its construction? The core of the answer may be given in a single word: efficiency. However, as we shall see, we will want to take this word in a rather different sense than that which ordinarily preoccupies programmers.

More specifically, the implicit dictions used in the language of mathematics, which dictions give this language much of its power, often imply searches over infinite or at any rate very large sets. Programming algorithms realizing these same constructions must of necessity be equivalent procedures devised so as to cut down on the ranges that will be searched to find the objects one is looking for. In this sense, one may say that *programming is optimization* and that mathematics is what programming becomes when we forget optimization and *program in the manner appropriate for an infinitely fast machine with infinite amounts of memory.* At the most fundamental level, it is the mass of optimizations with which it is burdened that makes programming so cumbersome a process, and it is the sluggishness of this process that is the principal obstacle to the development of the computer art.

These reflections suggest that some of the weight of programming be thrown off by passing from the programming dictions ordinarily used to a more highly "mathematicized" language.

Our hope to be able to make something of this general idea is raised by the observation that efficiency has two rather different sides. One is mathematical and abstract in character. What intermediate logical constructs must be built as a process proceeds, and how large are the sets of logical objects over which searches must be extended during such a process? The other side of efficiency is machine-related and basically two-fold. First, we must ask the question of inner loop efficiency: into what tabular representations can necessary abstract structures be mapped with advantage, and once this representation is established, how efficiently can the necessary coded processes be made to effect these tables? Then we must ask a fundamental question related ultimately to the speed chasm which separates electronic from electromechanical memories: How large are the data sets with which an algorithm will force us to deal? How can these data sets best be staged between different grades of memory so as to hurry the completion of an algorithmic process? We may remark that machine-related efficiency issues are apt to have as much or more to do with these memory management problems as with problems of inner loop coding, even though most programmers, especially those with an assembly-language background and bias, tend to think more of the latter. Our hypothetical mathematicized programming language would almost completely mask all machine-related efficiency issues. There is, however, no reason why it should hide those more abstract issues of process design which can easily have a more important bearing on efficiency. Indeed -- and this is one of the benefits for which we may hope -- it should, by masking the former, enhance our ability to concentrate on the latter.

The foregoing considerations lead one to suspect that a programming language modeled after an appropriate version of the formal language of mathematics might allow a programming style with some of the succinctness of mathematics and that this might ultimately enable us to express and to experiment with more complex algorithms than are now within reach. The notion of language that appears here then demands additional clarification. Speaking very general

a computer language is a set of notations referencing objects and processes, and satisfying all the following constraints:

1. A formal distinction between well-formed and ill-formed programs exists, and a "syntax checker" capable of administering this distinction can be built.

2. That class of objects and processes to which well-formed programs refer can be defined rigorously.

3. A "compiler" capable of transforming a well-formed program into the objects and processes that it represents can be built. These objects can in fact be represented within a computer, and the processes can in fact be carried out.

Since it refers as a matter of course to infinite sets, the language of mathematics has only the first two of these properties, not the third. Nevertheless, it is clear that in searching for a mathematicized programming language we will wish to start from some appropriate version of the language of mathematics. With which of the several variants of formal mathematics that might be contenders shall we begin? We choose to begin with set theory, formally represented, let us say, in its von Neumann-Bernays form. This is a language relatively free of artifice, close to the heuristic spirit of informal mathematics, and a formal system with which, in one or another version, there is a great body of satisfactory experience. In particular, we know that using the very small and simple set of primitivies that this language embodies that the whole structure of mathematics, from abstract algebra to complex function theory, can be built up rapidly, intuitively, and in a manner largely free of irritating artificialities. Taking this as our starting point, our problem becomes the following: Adapt set-theory to be machinable.

A development project sponsored by the National Science Foundation began at New York University in the Fall of 1970 and has continued up to the present date. Our project has concentrated on expressivity rather than efficiency as a language goal. We have felt that our somewhat unusual concentration has removed some of the obscuring underbrush that often surrounds the discussion

of fundamental issues in programming and has allowed us to comprehend certain issues more clearly than before. The target language of this project was designated as SETL (for 'set language').

Our work over the last few years has been embodied in a miscellaneous collection of semi-internal publications, the main items of which are as follows:

(a) A manuscript entitled "Abstract Algorithms, and a Set-Theoretic Language for Their Expression." This discusses certain general issues of programming language design; gives "users manual" information on a first version of the SETL language, and then presents a fairly wide variety of algorithms in SETL.

(b) A series, currently at No. 82, of miscellaneous studies and working papers growing out of the overall project. This series has appeared under the name 'SETL Newsletter'.

(c) A shorter early form of the manuscript (a), containing however a certain amount of material which was not repeated in (a) and which does not appear in the present manuscript.

Currently (summer 1975) a language ('SETLA') embodying a substantial subset of the intended SETL language has been implemented for the CDC 6600, and is in experimental use at NYU. Development of a fuller compiler yielding considerably more efficient code continues under way.

The material presented in the present volume is the second of three expected parts of an overall summary of work during the past several years on SETL, a new programming language drawing its dictions and basic concepts from the mathematical theory of sets. The general approach followed in this work, which has been carried out in the Computer Science Department of New York University, was presented in the first volume of a series of three, entitled Installment I: Generalities. The present installment focuses directly on the details of the SETL language as it is now defined. It describes the facilities of SETL, includes short libraries of miscellaneous and of code optimization algorithms illustrating the use of SETL, and gives a detailed description of the manner in which the set-theoretic primitives provided by SETL are currently implemented. A third volume, to be entitled Installment III: Extension and Optimization, is planned.

To have cited original sources for the numerous algorithms for which SETL codes are given in the present volume would have involved us in a considerable bibliographic effort, and we have not done so. The reader will recognize, however, that algorithms due to D. Knuth, John Cocke, Ira Pohl, Jay Earley, R. Floyd, and many other workers in various fields of computer science appear in the following pages. Those interested in tracing the algorithms which we give back to their original sources may consult the comprehensive treatise of Knuth, which gives extensive bibliographies and a careful historical account of many algorithms.

An effort has been made to achieve accuracy in the SETL algorithms given in the present work. Each of these algorithms has been read by several people; where implemented SETLB versions of the algorithms exist, the algorithms given have been compared with them. Nevertheless, it is to be feared that some bugs remain in our algorithms. Readers discovering such bugs are asked to send corrections to the author.

Currently (Spring 1973) a revised and considerably more efficient version of the presently available SETL subset language ('SETLB') is approaching completion. When running, this new version will permit a substantial increase in the level of experimental SETL use.

A comprehensive catalog of SETL-related material, including an up-to-date listing of the miscellaneous studies and working papers appearing occasionally as the 'SETL Newsletter', has been prepared. This catalog can be obtained by writing to

SETL Publications Coordinator
Computer Science Department
New York University
251 Mercer Street
New York, New York 10012

The work presented here is, like all computer work collective; all of it owes much to my collaborators on the SETL project. Mr. Henry Warren played a central role in the detailed design and coding of the 'run time library' of routines realizing the primitive SETL operations. David Shields has made numerous important contributions both to the specification and to the realization of SETL, as did Kurt Maly, Elie Milgrom, and Gray Jennings. We have profited in many ways from close contact with the elegantly realized BALM language of Malcolm Harrison, and from frequent technical discussions with Harrison. Useful criticisms of earlier versions of SETL, and in some cases extensive suggestions for its imporvement, were made by Jay Earley, Rudolph Krutar, Patricia Goldberg, and G. Fisher. Ken Kennedy developed many of the SETL otpimization algorithms presented in Item 16. The progress made to date in realizing SETL reflects the efforts of Aaron Stein, Bob Abes, Ed Schoenberg, Stephanie Brown, Edith Deak, and Samson Gruber. Kent Curtis of NSF has been an important source of general encouragement for our project, and has made substantial direct technical contributions to it. Milton Rose of the AEC had much to do with the inception and continuation of our work. I would also like to thank Robert Bonic for useful discussions concerning SETL, and Sam Marateck, Sheldon Finkelstein, Jerry Hobbs, Robert Paige, Kamal Abdali, Beatrice Loerinc, Max Goldstein, Henry Mullish, Aaron Tenenbaum, George Weinberger, Michael Brenner, and Peter Markstein for their participation in our work.

Item 1. ON THE SOURCES OF DIFFICULTY IN PROGRAMMING.

Programming is difficult, expensive, and highly time-consuming. That this should be so is surprising in view of the fact that a programming effort normally begins with what in most scientific disciplines is not a problem but its solution, namely, an overall algorithmic plan worked out to a convincing level of heuristic completeness. We see the explanation for this surprise as lying in certain general principles of complex constructions, principles which suggest certain views concerning basic strategic issues in the design of programming languages.

The development of a complex program, like the construction of any highly structured object, consists of a progression of steps that supply piece after piece of a total. For the total to be correct it is, of course, necessary that all these separate elements cohere correctly. Each element must therefore satisfy certain constraints. The set of all those constraints that affect the choice of a program element E may be called the *local context* of E. Note that in typical programming situations local context will be defined by a miscellany of restrictions, particularly the following.

1. Syntactic restrictions determined by the programming language being used and by any definitional extensions to the language that may be operative in a given context.

2. Semantic requirements reflecting particular properties of subprocesses (already defined or to be defined) that are to be invoked in a given context.

3. Semantic requirements related to the structure of the data objects to be manipulated in a given code section.

4. Accumulated odds and ends, as, for example, restrictions implied by the previous uses of particular data items, subroutine names, or so forth.

As noted, a program is built by choosing a sequence of elements, each correct in its local context. The probability that a given element E will be correctly chosen will fall off rapidly with increasing complexity of its local context, and, beyond a certain

threshold T of complexity, this probability will effectively be zero. The inverse of this probability measures the *difficulty* of choosing a given program element corrrectly, or, what comes to much the same thing, the number of iterations in debugging that will be required before the fully corrected form of an element is attained. In this connection it is useful to bear in mind the (merely suggested) shape of the difficulty versus context-complexity curve shown in Figure 1 below.

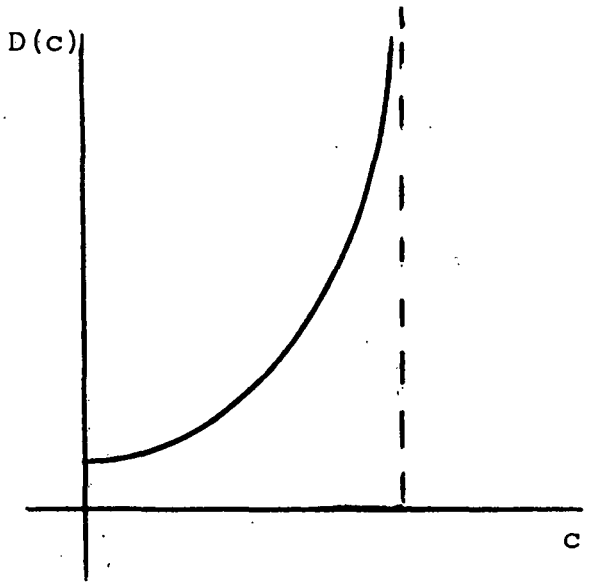


Figure 1. Local context complexity c vs. the difficulty D(c) of completing an item in compound structure.

The line of thought which leads us to the curve shown in Figure 1 may be extended to give an overall theory of the programming process.

Suppose that a total program P consists of elements $E_1,\ldots,E$ and that the local complexity of context of the element E is C. Then we suggest that the overall effort required to complete the entire program P will be measured by

(1) $$\text{tot}(P) \cdot D(C_1) \cdot \ldots \cdot D(C)$$

where D is a function growing rapidly with C and becoming quite large at some finite complexity threshold T. A formula of this sort can account for various observed features of the programming process, including the very large fluctuations that the quantity

tot(P) exhibits even when P remains fixed. (One group of programmers may complete a project many times more rapidly than another, even when both groups are involved with quite similar projects.) We take this to reflect the rapid growth of D(C) with C, a growth that would imply that relatively small increases in logical systematization, applied consistently, could have a substantial effect on the effort tot(P). The very large observed variations in individual programmer activity can similarly be derived from smaller individual variations in complexity tolerance.[1]

---

[1] The formula for total programming effort suggested above leads also to potentially useful insights concerning the development of a programming project during its lifetime. When an element E of a program is initially "sketched out," the logical complexity of its local context is momentarily elevated because various still-to-be-resolved uncertainties concerning undeveloped elements of P form part of the initial logical context of E. We call this transitional element in the context complexity of E *external irresolution complexity*. When the whole of a first draft of P is completed, this temporary contribution to local complexity disappears, ideally allowing the elements E to be confirmed (or revised as necessary) a good deal more surely and rapidly than when they were first elaborated. A contrary force arises, however, from the fact that various details specified during the development of elements $E_1,\ldots,E$ (and especially those relating to data structures) become part of the context of E. We call this contribution to the context complexity of E *accumulated external complexity*. The accumulation of external complexity may cause projects to behave pathologically, the context complexity of key elements actually increasing over the life of the project, which may make project completion impossible or at least very much more tedious than initially estimated.

These last remarks suggest certain principles that might be applied to determine the order in which the various parts of a complex project can most usefully be tackled.

1. The most complex elements of the project should be surveyed first, and the relationships of these elements to the remaining project elements (their "external environment") determined to some degree of approximation. Overall decisions concerning those aspects of the simpler project elements that form significant parts of the logical context of the more complex elements should then be taken. These decisions are to be made tentatively but should be sufficiently firm and detailed to relieve the more difficult elements of most of their external irresolution complexity.

2. Full program development should then begin *with the most complex project elements*, which should be brought relatively far along before detailed work on the simpler project elements is begun. Accumulated external complexity will then complicate only the simpler project elements which will have a less harmful effect than increases in the context complexity of already complicated elements. [continued on next page]

The view of the programming process embodied in formula (1) leads to the conclusion that a main aim of any programming language design must be to provide tools which make it possible to describe the whole of a desired totality in a maximally modular fashion. That is, we need to invent mechanisms capable of preventing the propagation of complexity between sections of an extensive program.

Here however a basic difficulty arises. By making a program highly modular, which implies the wholesale use of standardized data structures and standardized rather than specially tailored process-patterns, we commit ourselves to a style of programming which may imply certain very substantial inefficiencies. A modular style of programming hides away many important internal process details which could be exploited by a (purely hypothetical) all knowing and perfectly accurate hand programmer, uses inefficient general routines in situations which a hand programmer could recognize the use of much more efficient special sequences to be appropriate, and so forth. It is this fundamental difficulty with which the language designer struggles, attempting to invent dictional patterns which are highly expressive and yet can be compiled into reasonably efficient running code. We may represent the general outline of the relationship between linguistic modularity and efficiency by the "tradeoff curve" shown in Figure 2 below.

As shown in Figure 2, the more we are willing to complicate a program by the addition of efficiency-enhancing special devices, the more we can expect its run-time efficiency to improve (provided, of course, we neglect the effects of errors in judgement). Conversely, the more we insist on extreme simplicity in the statement of an algorithm, the less we are able to guarantee efficiency.

---

[1] [continued from previous page]
The above considerations underscore the importance of braking a complex project into smaller relatively independent pieces and of staging the project in a structured manner that allows the treatment of potentially complicating factors to be postponed when possible. Linguistic mechanisms that accord with this intent will be used in the SETL language.

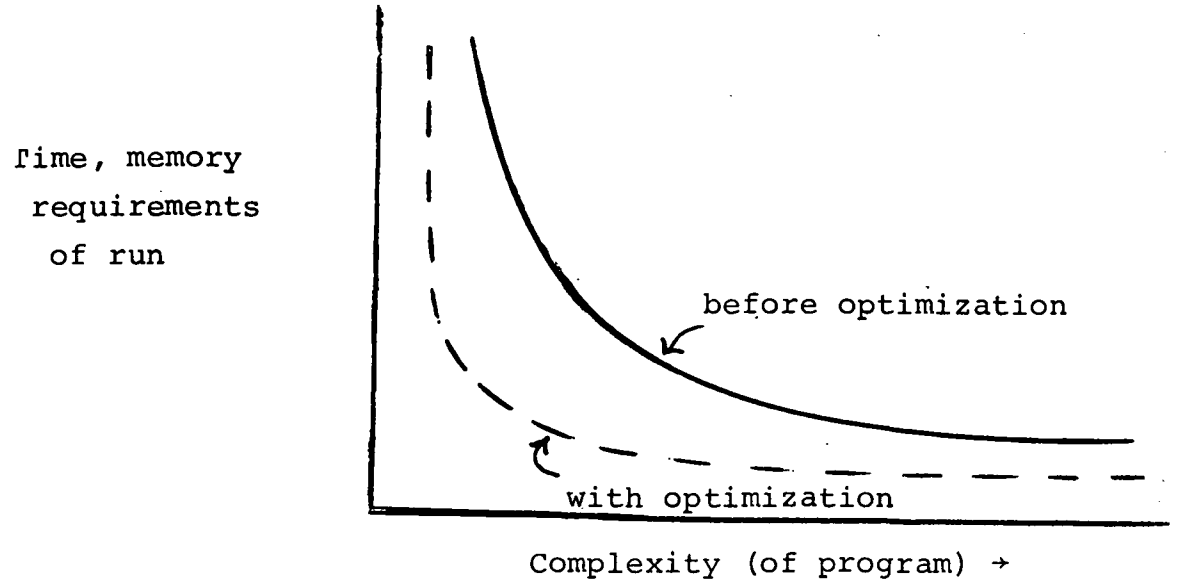


Figure 2. Tradeoff between Program Complexity and Efficiency, Showing the Effect of Optimization.

If, hwoever, we find ourselves able to construct an automatic optimizer which, after analyzing the text of an algorithm, can transform it, producing a more efficient text incorporating many of the efficiency-enhancing special devices which a hand programmer would normally use, much potential loss of efficiency can be avoided. Improved optimization techniques therefore form an essential part of an overall attack on the problems of programming, In every case, however, the language designer will explicitly or implicitly choose a point on the tradeoff curve of Figure 2 as the target of his design. That is, he will decide on the extent to which efficiency of target program is to be sacrificed in favor of simplicity of program text. In this functional sense, SETL aims to be a language of the highest degree of expressivity consistent with even that modest degree of run-time efficiency needed to support any amount of actual computer usage. Formalized mathematics itself is of course the language that results when even this modest demand for efficiency is dropped.

We have therefore as our target a language intended to be of maximum expressivity in the sense just described; this will be a language in which programs are built of a small number of powerful lements fitting together in a manner governed by uniform simple conventions, rather than of a great many microscopic elements relating to each other in complex and irreulgar ways. Drawing

from earlier discussion the principle that lack of modularity is the main danger which the use of such a language is to avoid, we must next ask: what are the principal forces which when programs are constructed tend to increase the degree of interrelationship between their elements?

We locate the most substantial of these forces within the logical linkages which connect the separate processes invoked within a program to the data structures to which these processes refer. Since processes in procedural languages are transient but their effect on data objects carries forward, process definitions will normally not exert so pervasive a force as data structure definitions in propagating complexity. Of course, if inappropriate subprocesses are specified as standard, this can propagate complexity to remote contexts. Nevertheless, really virulent increases in the local complexity of a program will normally be traceable to data structure related sources. Thus, to hold the complexity of local context within a program below a fixed limit, we must, more than anything else, standardize the data structures that a program and a programming language use.

The use of standardized data structures will imply a certain standardization of the processes that manipulate these structures. In particular, we will find it necessary to avoid the use of processes that can create structures of non-standard form. This restriction concerning the processes to be used has then significant implications of its own. If we ever make use of operations that transform these structures into nonstandard forms, the nonstandard details of these forms become part of the logical context of every instruction that might manipulate a nonstandard structure. Complexity can build up very rapidly in such situations, and for this reason we will prefer to avoid them. Thus, once having chosen certain data structures for standard use, we will normally proceed at once to standardize a family of basic operators addressing these structures and to describe compound processes only (or almost only) in terms of the set of basic operators thus designated. The approach sketched here, which makes use of standardized data structures and of combinable basic

operators affecting them, is that which lies at the basis of every programming language design. The standardization upon which we insist can, of course, have a negative effect on the efficiency that our method will attain in any particular case; and it is this consideration that may again and again tempt us to the use of more highly varying nonstandard forms. Within the approach envisaged here, howver, efficiency enhancing nonstandard variations are allowable only in a phase of programming subsequent to the initial layout, in terms of standardized elements, of a programming approach. By relegating efficiency related design supplements and redesigns to a second stage of programming, we confine, to a limited set of specified contexts, a significant class of programming activities likely to propagate complexity. In particular, we become better able to avoid inadvertent decisions in particular contexts which propagate complexity to other portions of a program, decisions of this kind can, when higher level preplanning is absent or insufficiently detailed, have most unfortunate complexity propagation effects. Moreover, by thinking first in terms of a standardized class of data structures, and only subsequently of those concrete variations that are truly desirable for increased efficiency, we will often find that the range of variations that must be made is smaller than could have been realized at first, so that these variations can themselves be standardized. In such cases, we will be able to develop automatic optimization methods that incorporate efficiency-enhancing variations in programs written in languages of high level and that produce programs that compare favorably to those developed in lower level languages by programmers forced to function in contexts straining their maximum complexity tolerance.

A data structure S in which many processes interface may itself tend to grow complex, and we will therefore wish to prevent its full complexity from affecting, to an unnecessary degree, the various processes that must address it. We therefore wish it to be possible for each process P to deal only with those aspects of S that are of concern to P, ignoring all others. That is, each process P should be able to view S in whatever logical "projection"

is appropriate. To allow this we will wish P and S to be linked via *access functions* that allow P to reference those aspects of S that it must. Note in particular that we find it desirable to represent the necessary access operations explicitly, rather than to represent them implicitly in coding patterns to be used throughout P. In particular, those basic attributes of S to be accessed and modified ought to be explicitly named within P and not represented in implicit fashion by compound access sequences; mnemonic names, rather than numerical subpart addresses, should be used for attributes, etc.

The emphasis that we have placed on the role of data structures in determining certain of the fundamental properties of programming languages that address them suggests certain further points. A programming language ought to incorporate powerful methods for the definition of compound data forms and for the specification of new operations applying to and combining them. Explicit mechanisms that allow operators to be related to the data structures on which they are to act should be provided. The ordinary semantics of assignment statements needs to be generalized considerably, allowing structures just as general as those which appear on the right-hand side of assignments to appear also on the left. We will also wish to develop a special declaratory extension of our algorithm oriented principal language; this extension will enable the "data strategy" (encoding, packing, access paths) to be applied in realizing a given algorithm to be specified declaratively in a succinct and centralized way.

In addition to the data related issues treated above, various other ways in which language design can aid in limiting context complexity may be noted. Note, first, that by making decisions in separated stages whenver possible, solving initial parts of a problem without foreclosing possible approaches to the parts which remain, we can reduce the complexity level with which we must deal at any given time. This may be called the *principle of decision postponement*. In decision postponement lies one of the basic advantages of the use of specification languages and

the "two stage" programming style to be discussed below. Various more direct hints concerning language design may also be drawn from this principle. First, a language should be able to treat in a syntactically identical way semantically analogous entity classes which may be substitutable for each other, so that the decision concerning which type of entity is actually to be used can be postponed. This implies also that we will wish to be able to postpone the choice of the detailed encodings and data layouts to be used in realizing an algorithm until the soundness of the algorithm has been verified.

The inner details of a program section should be isolated to a maximum degree from detailed conventions determined by other program sections. This may be called the *principal of structural isolation*. Note in particular that semantically unitary items established outside a program element E should be represented within E by unitary names and not by complex sequences defined by external program sections.

Those items to whose details a given item E is most closely related should find places physically near E without distracting less closely related material being included. This may be called the *principle of grouping by logical relation*. In accordance with this principle we find it undesirable for a language to establish rigid conventions concerning the order in which syntactic elements must appear. In many situations, it will be useful for program text to embody a 'footnoted' style, with the main outlines of process and flow being shown in a "lead paragraph", and with details which flesh out these outlines following. An adequate language will provide for a variety of linguistic styles providing textual clarity in a variety of logical situations, and will incorporate powerful extension mechanisms allowing a user to develop significant personalized language features. The language extension tools to be provided must be adequate to deal with those common situations in which masses of detail must be repeated with obligatory small variations because of language problem mismatch. But these tools should also allow profounder global

transformations of a source text, expansion of the basic semantic object classes which a language recognizes, and growth in the family of declarations which a language provides.

Specialized control and linkage structures may also aid in increasing the power of a language to deal with certain important classes of situations. "Whenever" dictions of the type used in many simulation languages are also quite useful in describing certain classes of algorithms. Various special types of plan-forming algorithms can be expressed most naturally in a language allowing "non-deterministic branch" dictions. Some of the problems of module linkage, subroutine naming, etc. which arise in connection with large programs are ameliorated if special linking dictions, providing for a more highly centralized control of subroutine naming and linking than is ordinarily used, are available.

We may note in conclusion that no language design is entirely complete until the debugging tools to be made available to a language user have been thought out. This is a matter often neglected in current practice. Presently, the sequence of events in debugging is typically as follows:

1. Anomalous program behavior indicates that an error exists. Direct consideration of the error by a knowledgeable programmer will normally point the finger of suspicion at some more or less restricted section of a total system or program.

2. The code section which has fallen under suspicion is taken up for examination. When originally composed, this section was considered, on the basis of an informal set of mutually supported programmer assumptions concerning its action and the moment to moment state of the data structures which it uses, to be correct. Certain of these assumptions will have been verified in particular cases by test runs. However, both original programmer assumptions and test history will normally have been lost when a bug appears. It will therefore become necessary to reconstruct these vital logical assumptions from the program as it stands, using for this purpose whatever disjointed or systematic indications of intended logic the program text happens to contain. Note that this reconstruction may have to be performed many times, often from comments, that are quite fragmentary.

3. Once reconstructed, the original programmer assumptions concerning the way the program should work will be reverified, using a mixed assortment of generally manual techniques. Those few assumptions which seem to play a key role, or which have come under suspicion, will be spot-checked. In debugging, one will find it necessary to explore an often very large program event space using tools that are generally quite weak. The program event space to be explored will be particularly large in those cases in which a compiled code lacking all information concerning the intended meaning and origin of bit patterns runs past an original fault for thousands or millions of instructions before encountering an error which it can recognize as such. One may also be required to scan large amounts of source text, without useful programmed tools being available for this purpose.

Debugging tools which can alleviate the deficiencies which have been depicted can be designed and should be a part of every language system. A means should be provided by which programmer assumptions concerning the functioning of a program can be stated explicitly and can remain an integeral, permanently maintained part of a program text, capable of being switched on and off in stages as debugging is pursued now and again at different levels. All the "assumption statements" generated during the debugging history of a program should be retained in appropriate forms in its text, so that they may be systematically reverified if the text is modified and debug mode is reentered. A powerful "program event" oriented language should be provided for the dynamic detection and rapid isolation of run-time events of debugging significance. An interactive scan language making it easy to gather together whatever fragments of source text become important at a given moment of debugging should also be provided; this should allow relevant passages of text gathered from all the parts of a large program to be displayed together. Finally, a language should include an appropriate and extendable family of "meaning declarations" which, by supplying more information to a compiler than can be gathered immediately from a bare program text, make various compiler administered static and dynamic consistency checks possible.

Item 2. A SECOND GENERAL REFLECTION ON PROGRAMMING.

What is programming? I will elaborate a series of answers to this pregnant question.

1. *To start with, programming is the activity that builds the interface between man, on the one hand, and computers, on the other.* Certain of its characteristics will then be determined by man and others by the computer. The goal of programming is the construction of advanced function, which requires the perfection of complex programs. Therefore

2. *Programming is the process of constructing complex objects.* In the preceding pages, certain basic laws affecting such processes of construction were outlined. To repeat, compound objects are built by successive correct choices of a sequence of elements $E_1,\ldots,E$ each element E must be chosen in a logical context that summarizes all those aspects of other elements that are relevant to the choice of E. We call the collection of all these influences the *local context* of E, and call any reasonable numerical measure of this collection the *context complexity* of E. It may then be observed that the chance of choosing E correctly falls off very rapidly as its context complexity increases, and effectively becomes zero at a not very large threshold T. This observation allows us to define the class of *constructible objects:* an object is *constructible* if it can be built by choosing elements successively, each in a context of complexity less than T. A function is *programmable* if it can be realized by a program that is constructible.

To construct a large object successively, one must therefore combine many subelements. The rules according to which elements may be combined are, of course, part of the logical context of every element. These rules must therefore be simple. But a simple set of rules allowing the indefinitely iterated combination of simple elements into a large totality defines some sort of "algebra." Therefore

3. *Programming constructs compound objects from simpler elements by combining elements according to the rules of some "algebra".*

To program, therefore, one must be aware of some such algebra, which must be capable of generating objects representing useful processes. Before they can be used, such algebras must be found. We conclude therefore that in a deeper sense

4. *Programming is the discovery of algebraic principles allowing the iterated combination of elements into compound objects representing useful processes.*

Next, observe that although the maximum threshold T of tolerable complexity postulated above will vary from person to person, for no one person is it very large. In this regard a group of people is no better than a single person. Therefore an object not constructible in the above sense can really never be constructed directly, either by individuals or by large teams. And it is very unlikely that such an object will be formed spontaneously by the action of a random process, even if this process acts repeatedly over long periods of time. Objects irreducibly unconstructible must therefore remain nonexistent. The barrier to their existence should be as firm as those set for mathematics by theorems of the type of Gödel.

There is, however, a way in which we can hope to find a way around the obstacle revealed by these pessimistic reflections. To see this, observe that the maximum context complexity of the elements of a compound object is by no means independent of the representation of the object. What in one representation may appear as a densely interconnected mass will in another representation appear as an object, perhaps still large, but consisting constructibly of items no group of which are impenetrably related.

To discover this second representation of a programming problem is to break the problem's back, since this discovery allows one to build what formerly were obscurely integral objects using systematic incremental techniques, that is, to proceed by the progressive accumulation of tables of information possessing no overwhelming degree of internal interconnectedness. In a still higher sense, therefore,

5. *Programming is the discovery of viewpoints or logical transformations that uncover hidden algebras in terms of which compound objects representing useful processes may be built. That is, programming is simplification, and, like mathematics, is a hunt for lucky simplifications.*

It is worth emphasizing that the discovery of these simplifications is the essential goal of experimental, as distinct from applied, programming. If in a strictly research situation we build a highly compound object, we do so only in the hope that immersion in the realities of a particular construction process may put us in mind of principles allowing this process to be simplified.

The transformation of a constructible compound object into that more highly interwoven form in which it directly represents some interesting function plainly amounts to a kind of *compilation*. (The practical possibility of carrying out such transformations is, of course, the contribution of the machine to the process of programming, which, in the preceding remarks, we have viewed almost exclusively from the human side of the man-machine interface.) We may therefore say that

6. *Programming is the discovery of algebras allowing the construction of objects worth compiling and is the programming of compilers for these objects.*

Elements that programmers are to combine need to be simple externally. But, as long as their internal complexity can be hidden, they need not be simple internally. Indeed, when objects having simple external description but embodying powerful function can be allowed within an organized algebra, the programmer's reach is multiplied. Hence

7. *Programming is the discovery of highly functional logical entity types possessing simple external descriptions and thus capable of being integrated into an algebra useful for the construction of still higher functions.*

The above remarks predicate an indirect method for creating functioning machine-level process representations. Our reflections concerning context complexity suggest that in the construction of

highly compound objects such an indirect approach is inevitable. owever, since this approach is, to begin with, fixed upon simplification and standardization as goals, in following it we run the risk of ignoring alternative constructions that might realize a given function in a particularly efficient way. Efficiency-oriented departures from a standardized approach are traditionally the prerogative of skilled human programmers. The mind, ranging analytically, can incorporate very useful variations into a basic approach, as long, that is, as the additional complications that such departures cause do not carry one over the threshold T of allowable context complexity. The programming range that we contemplate will, however, involve transformations of form so repeated and elaborate as to exclude the possibility of external meddling with the compiled versions of objects. Given that we will have to allow efficiency-enhancing variations to enter into the compilation process, it follows that in the programming range we contemplate it will be found necessary to systematize these variations and to build a *program* capable of weaving them into the compiled version of an initial text. Such a program must, of course, be able to analyze programs in sophisticated global ways. The programmer may assist this optimizer by adding, to a text to be compiled, disjointed declarations that summarize and transmit significant conclusions concerning the text, but his role may not safely be allowed to exceed this limit. We may in this regard say that

8. *Programming is optimization, that is, is the programming of optimizers able to analyze and improve other programs and is the discovery of principles that allow the simplification of such optimizers.*

The use of the indirect technique suggested above, involving the optimizing compilation of sequences of constructible objects, will eventually allow functions that lie utterly beyond the scope of more primitive direct methods to be programmed. Nevertheless, just as Gödel's theorem assures us that certain rather simple questions lie quite out of the range that the method of mathematical proof an reach, so we may also take it that certain functions that might

be of great use are not programmable in that no constructible object can represent them, even after compilation. It is therefore of interest to consider whether the construction of artificial intelligences is at all possible. Might it not be that, among all those objects constructible within the maximum complexity threshold T of the human mind, none exists that can represent all the capacity of the mind?

In coming to grips with this question, one must first of all realize that it concerns innate and not learned capacities. That which is learned is drawn from an accumulation of separately encountered facts, presented in no particular order or relationship. No inextricably interwoven object is immediately represented in the pile of fragments presented as input to the learning process. If facts within the mind are interwoven in uncompilably complex ways, they can be so only because the mind is innately capable of establishing exceedingly complex connections. If the ability to learn can be programmed, the teaching process will be trivial. That which we seek to duplicate is therefore as fully present in the neolithic savage as in the *savant*.

But might not this innate facility, in spite of the somewhat restrictive definition that the above remarks give it, still be unprogrammable? It might. But I doubt that it is. Hard evidence in this area is still missing. To argue from what has not been done, or from the collapse of inflated initial projections, is an absurdity, given that the computer is still less than twenty-five years old. It seems to me that the fragmentary evidence that does exist ought to incline one rather strongly against such arguments. Substantial progress toward the programming of mental function has been made in a few cases. For example, the parser-compiler type of program captures a striking part of the ability to learn languages. Note that, in accordance with the general principles stated above, it is the discovery of an underlying algebra, specifically the algebra of pattern combination in the manner embodied in BNF grammars, that enables us to construct such programs.

One may conjecture that mental faculties that, like the ability to learn languages, are generalized and involve explicit learning

will prove to be more readily constructible than faculties, such as visual pattern analysis, that are more rigidly fixed. Learning at the level of language learning is surely of late evolutionary arrival, and one may therefore surmise that this faculty has not had the time to grow as complex as have others. In view of the general pattern that evolution exhibits with regard to physical organs, we may take another hint from this observation. Speech and higher reasoning, recently evolved, may possibly employ specially adapted versions of faculties that antedate them. If this is true, then successful duplication of the mind's language-handling faculty may provide clues valuable for the analysis of still other mental functions.

The optimistic remarks of the preceding paragraph, if they can be trusted, lead one to try to put the question of artificial intelligence quantitatively. The programmability of a complex function is, as we have seen above, defined by the battery of simplifying transformations that determine one's programming technique. How many as yet undiscovered simplfication principles remain to be found before artificial intelligences will, in this sense, become programmable? If and when these principles become available, how large a body of compilable text will be required to define the intelligence? I emphasize again that the text in question is that which organizes the intelligence's capacity to learn, not that possibly larger body of text that defines the total mass of facts available to it. That is, an intelligence is defined by those highly integral programs that determine the principles according to which it organizes more disjointed information tables subsequently fed to it. It would be rash to try to answer the questions just raised. Nevertheless, putting them serves, when one notes the extent to which a simple yet well-organized programming system such as LISP makes it possible to define quite striking language processing faculties by quite a small body of text, to buttress optimism. Putting these questions also serves to emphasize the central importance, for the eventual construction of artificial intelligences, of progress in programming technique. They also tell us what to look for: transformations that allow originally integral

functions to be represented incrementally and in this sense to become learnable. Thus, for example, we may recognize that the organization of at least part of the language-analysis function around an explicit Backus algebra of syntactic patterns is a very significant step, the sort of thing that we must energetically seek to extend. Other functions can be cited for which organizing "algebras" are desirable and might be possible. An associational "feature noticing" function of a generalized sort would be useful in a wide variety of situations, for example in optimization by the method of "special cases," where such a mechanism might permit the easy addition of new optimizations. At a more technical level, a language of memory management, allowing certain central problems of concrete algorithms to be treated systematically, could enhance our ability to produce efficient versions of concrete algorithms rapidly.

In connection with this last remark we may raise yet another quantitative question concerning artificial intelligence. The capacity of an intelligence is measured both by the level of function that its responses embody and by the speed with which these responses can be generated. Assuming that it becomes possible to construct an intelligence, how fast will this intelligence be able to think? This question touches on all those questions of efficiency that, by concentrating on abstract programming issues in our preceding remarks, we have neglected. Its answer will, of course, be determined both by the basic capacities of the hardware available at a future date and by the extent to which optimization is able to overcome the natural tendency to inefficiency of a highly compiled programming style. Until now, almost all the most dramatic increases in program speed have come from basic hardware speedups. In a few cases, as with the development of the fast Fourier transform, fast sorts, hashing, and list-organized search techniques and the improvement of certain little used combinatorial algorithms, programming has made similar contributions to efficiency. The domination of efficiency by hardware should continue for at least a while longer, as clock cycles diminish toward 10 nanoseconds and especially

as improved manufacturing processes weaken the I/O barrier by making greatly expanded electronic memories available. In this regard programming may for a while have the largely subsidiary role of choosing algorithms that bypass potential combinatorial disasters. A more systematic but perhaps less immediately significant contribution of programming to efficiency will probably come through the continued development of optimization methods, especially those that, like cross-subroutine optimizations, aim at preventing the efficiency losses that a naive and highly compiled programming technique would imply.

Efficiency loss through the use of such techniques is in fact far from being a crucial problem. It has generally been true that, once able to organize a given programming area clearly, one has also been able to invent systematic optimizations that permit indirect programming techniques to attain an efficiency comparing not badly with results obtained by the use of much more expensive and eventually quite impractical manual techniques. In regard to the programming of intelligence, it may also be remarked that, once we are able to create a faculty, we may expect to be able to improve its efficiency substantially by providing it not in the most general form possible but in a specialized, "reflex-like" rather than fully "adaptable" form.

As the simplifying techniques needed to organize complex functions are progressively revealed through the progress of programming, the significance for efficiency of those elemtnary subprocesses exercised most constantly by the compiled form of programs written using these techniques will become plain. By realizing such "inner" subprocesses in hardware, one improves their efficiency through the elimination of unnecessary generality and by that use of large-scale parallelism that gives such great advantages to hardware realizations. An example of the type of situation we have in mind is currently seen in the tendency to simplify programming by speaking in terms of extremely large "virtual" memories. Such an approach makes constant use of certain simple "memory-mapping" operations and has led to the construction of these functions in hardware. Similar future influences of programming concept on hardware design are to be expected.

Artificial intelligences, if realized, will take programming as one of their first tasks, and it is interesting to try to guess the effect that this might have on programming. One of the great advantages of such intelligences will be their enormously large complexity tolerance, as compared to the capacity of the natural mind. In connection with the remarks made above we surmise that this will greatly extend the class of programmable functions, though in what way is not clear. Certainly, however, they should be capable of optimizing programs to a degree impossible to the natural mind and in this way can contribute substantially to the improvement of their own efficiency.

Item 3. ADDITIONAL GENERAL REFLECTIONS ON PROGRAMMING.

I. What constitutes progress in programming?

Donald Knuth has called programming an 'art', and has argued the appropriateness of this designation at some length.[1] In this short essay I should like to argue (though of course terms are not necessarily matters of great consequence) that programming is not an art but a nascent science. The distinction that I see is this: art, though ever changing and fresh, does not and cannot progress, since it lacks any real criterion of progress; but science does progress.

To establish programming as a science is therefore to propose a convincing criterion of progress for it. To this end a comparison with mathematics is enlightening. Mathematics is the search for interesting proofs, and for general frameworks which allow interesting proofs to be found. A proof is defined by its target theorem T, but nonrecursively; even after T is conjectured (which may itself be a significant event) its proof can be arbitrarily difficult to find. Thus the moments of progress in mathematics (typically they are discrete and sharply defined) are (simplifying somewhat) the moments at which proofs are found. Note also that once T is proved, and assuming that T is truly interesting, it will illuminate some broader area, and in particular will ease one's approach to other interesting theorems.

There certainly is a side to programming, namely the invention of algorithms meeting efficiency constraints whose satisfiability is nonobvious, which has just this flavor, and which is therefore as much a science as mathematics. (Knuth is of course one of the main developers of this 'single-algorithm' oriented part of programming science). The Fast Fourier Transform is no less an invention than the Pythagorean Theorem. But should the other side of programming, namely its integrative side, i.e., the growing collection of techniques used to organize large systems

1 See Knuth, Computer Programming as an Art, CACM 17, 11 (November 1974), p. 667.

of algorithms into coherently functioning wholes, be considered as an infant science also, or must it remain an art?

I argue that this part of programming is a science also, albeit a science only in its infancy. To see that it is, one must observe that the crucial obstacle to the integration of systems of programs providing very advanced function, which will generally be large systems of programs, is met when their complexity rises above the very finite threshold beyond which the mind can no longer grasp them totally. (This point is developed at greater length in Item 1 above.[2]) Those who have had the experience of working with systems of this level of complexity will realize that one's ability to cope with them is quite limited, and always threatens to founder entirely. With the active help of a computer, by assembling multiperson groups (less prone to fatigue than individuals), and by concentrating on one system portion at a time one can manage such systems. But even while being successfully developed and maintained they remain elusive and largely inexplicable; in a manner never fully comprehended or controlled, they evolve. In contrast, a system which remains below the threshold critical for full comprehensibility can be designed with assurance and implemented with a firm grasp.

Thus programming progresses when schemes which make it possible to realize significant function without overstepping this critical threshold are invented. Each such scheme will address some more or less broad application area, and will provide objects, operations, and also a semantic framework within which these objects and operations can be combined together into large structures, the whole allowing significantly many functions which formerly would have required superthreshold realizations to be written out completely without the critical threshold of complexity being crossed. Proof of the success of such a scheme comes when, by approaching a major application in a way conforming

2 The same point is also central to Dijkstra's essay Concerning Our Inability to Do Much, p. 1 in Structured Programming, O. J. Dahl *et al.*, Academic Press, 1972.

to the rules of the scheme, one finds that it has become comprehensible, though it was not so before. A framework of the kind envisaged is of course a language, and another proof of its success will lie in the fact that this language allows one to speak clearly and directly about important matters which previously could only be depicted in roundabout and clumsy ways. (In mathematics, major definitions have the same effect.) Note also that the restrictions which such a framework embodies can, if they prevent complexity from rising rapidly, be just as important as the flexibility it provides.[3]

Once such a framework has been invented, and when some process or function has been specified in it, it will generally not be hard, though of course it may be tedious, to take this specification and transcribe it, perhaps to gain efficiency, into some available and appropriate programming language. Because numerous errors are bound to infest any lengthy or complicated process of transcription, it is generally useful to implement languages which realize the framework or something close to it in as polished, succinct, and helpful a form as possible. Among other things, this can call for the development of elaborate program program analysis methods, which for example may be used to support rich systems of explicit or implicit declaration, to provide sophisticated diagnostics, or to perform optimization which the user of a language of very high semantic level is expected to omit. But such development is tool-building rather than fundamental progress. In this sense, I consider that SNOBOL and SIMSCRIPT, for all their lack of polish, embody very significant inventions: SNOBOL the string/pattern algebra and a natural framework for organizing operations in that algebra; SIMSCRIPT the event and scheduling notions so helpful for simulation. Similarly I would say that the interest of ALGOL 68 lies not in its syntactic polish, but in the way it handles object types and coercions, and in the fact that the kind of systematic approach to declarations which it embodies promises to reduce levels of run-time error very decidedly.

This point lies at the heart of Dijkstra's celebrated note Go-To Considered Harmful, CACM 11, 3 (March 1968), and of various of Hoare's interesting comments on programming technique, e.g., Monitors: An Operating System Structuring Concept, CACM 17 (October 1974) p. 261.

II. What programmers should know.

It is now useful to recast the views concerning the programming process which grow out of the point of view developed in the preceding pages, formulating these views as recommendations concerning the intellectual equipment and cast of mind which a creative, high level programmer should attempt to acquire. We have in mind programmers (or designers) who originate programs, rather than programmers (alas! the vastly more numerous group) whose work is the extension and repair of programs poorly done and documented in the first place, and the adaptation of these programs to shifting system interfaces. And we will stress the 'higher' rather than the commonplace aspects of the programmers' intellectual armament.

A programmer should understand:

1. *Algorithms*, i.e. various important algorithmic inventions using which significant processes can be performed with special efficiency. Examples are heapsort, fast Fourier transform, parsing techniques, fast polynomial factorization methods, etc. He should understand that a formal concept of program performance exists, and have some familiarity with the combinatorial techniques used to analyze algorithm performance. In this connection, it is also important to understand that there exist processes which no program can carry out rapidly, and others which no program can carry out at all.

2. *Semantic frameworks*, which allow individual algorithms to be organized into large program structures. He should understand the use and significance of such fundamental semantic inventions as subroutine linkages, space allocation, garbage collection, recursion, coroutines, and various structures useful for organizing processes acting in parallel. He should be familiar with object/operator algebras which are of general significance or which play an important role in significant application areas: sets and mappings, strings and patterns, Curry combinator and lambda

calculus, etc. He should understand the way in which semantically significant languages make these frameworks and algebras available, and the way in which the syntactic features of a language facilitate the use of its underlying semantic capabilities.

3. The programmer should have a conscious view of the *programming process*, understand the way in which programs, in their earliest origins, coalesce out of less organized intellectual structures, and understand the objective/psychological influences which can either facilitate the development of a final, efficient and reliable program version or abort this development. Accumulating complexity should be understood as a central peril to successful program construction, and techniques for managing and minimizing this accumulation should be appreciated. Particularly important among these techniques are the orderly multilevel development of more and more efficient program versions through a sequence of progressively less high language levels, and also prespecification, for each major application, of a well-tailored set of application-specific primitives, expressed as macros, structure declarations, or auxiliary subroutine definitions. Simple clean logical structure should be perceived as a central goal of programming; and each simplification seen as a victory, each complication as a defeat. The programmer should learn to structure his programs in spare, logically clean ways which keep open the possibility of subsequent functional expansion.

4. The step which leads from a high-level program representation to a lower level and more efficient version of the same program should be seen and approached as a process of *manual optimization* to be carried out in a mechanical spirit. For use in this process, the programmer should have knowledge of a wide variety of optimization approaches and optimizing transformations, adapted to the various language levels at which optimization will be directed, and ranging from high level global program restructurings to machine level inner-loop bit-tricks.

5. The manner in which the global properties of an algorithm determine the *data structures* appropriate for the representation of the objects which it manipulates should be understood. The programmer should have a wide variety of data structures at

his disposal, and understand the efficiency with which these structures can represent more abstract data objects and operations.

6. The fact that very small *inner loops* are often critical for program efficiency, and that conversely most of a program lies outside its efficiency critical paths, should be understood, which implies that it is important to measure actual program behavior before committing to the optimization of any particular section of code. (Note that the optimization of large non-critical program sections represents an unwarranted expenditure of program resource.) He should be familiar with the tools for measuring program behavior which various languages, operating systems, preprocessors, and program editors provide.

7. The programmer should understand the techniques which can be used to adapt programs to run well in specific *operating environments*; this implies knowledge of data staging, overlay, paging, and virtual memory techniques. The principal factors which affect program performance in these environments should be understood, as should the way in which programs can be structured to isolate environment dependencies and preserve interenvironment portability.

8. The *correctness of a program* rests on a web of logical relations, implicit in and guiding the program's development; this set of relationships, if made manifest and formally complete, would constitute a formal proof of the program's correctness. An essential part of program development is to guard the integrity of this web as successively more specific program versions are developed, to structure programs so that the logical assumptions on which it rests do not become unmanageably complex, and to check the logical integrity of the program systematically and repeatedly as it is developed. The fact that some programming language constructs aid in the preservation of logical integrity, while other more dangerous tools tend to tear a program's underlying web, should be appreciated.

The process of debugging is that of searching, in the possibly very large execution-event space of an ill-behaved program, for *primary anomalies*, i.e., places at which good input leads immediately to bad output; these are the events which point to program errors. The debugging tools which make it possible for this large space to be searched should be mastered; bugs should be recognized as inevitable and programs prepared in ways which facilitate their detection and removal; but debugging should be seen as a process for the repair of a relatively small number of tears in an extended and delicate logical fabric, rather than a process which can bring order into a heap of disconnected strands. During program debugging, the programmer should always understand the degree to which the tests which he has administered 'cover' all the possible lurking-places of bugs, and should design tests systematically for maximum coverage. The types of program constructs likely to give rise to bugs, and the types of bugs typically to be expected, should be understood, and the kinds of static and dynamic consistency-checking likely to uncover bugs rapidly understood also.

Finally, the several techniques of formal program-correctness proof should be known, and the implications of these techniques for the construction of relatively bug-free programs and for bug detection comprehended.

9. In conclusion, we list the various important *hand skills and habits* of an elementary but important sort which the programmer should have. He should know the interactive, editing, and program maintenance aids available to him; program carefully, check conscientiously, and document scrupulously, always remaining aware of himself as a team member whose expensive product must reliably serve others. He must realize that programming is a highly unstable process, in which a disorganized effort can consume ten times, or even a hundred times, more resource than a well devised effort with the same goal, and that especially in programming, work is a *signed* quantity, and mere activity, no matter how energetic, is no proof of significant contribution to a goal.

Item 4. ON THE UTILITY OF AN INEFFICIENT SPECIFICATION LANGUAGE.

We have suggested in the preceding paragraphs that a programming language considerably more mathematical and expressive than those now in use can be designed, and have implied that the development of such a language must be a central element in any attack on the present problems of programming difficulty. The increasing disdain for new language that some of the most sophisticated computer scientists have expressed of late leads me to put the following question very directly:

Is it reasonable to expect that the definition and implementation of such a language will, merely because its mathematicized character and systematic adaptation to the purpose of algorithm specification, create any advantages at all? I contend that the answer is yes, and I contend, morever, that the benefits that such a language will provide are numerous and substantial. I buttress this claim by indicating one of the most general of the beneficial effects expected, an effect upon computer science education. The availability of a mathematicized programming language should in relatively short order lead to a restructuring of the way that computer science is taught.

Presently one begins with a basic course, titled variously but generally called something like Introduction to Data Structures and Algorithms. When an appropriate abstract specification language becomes available, this fundamental subject matter should fall into two parts, which might be separately called *abstract algorithmics* and *concrete algorithmics*. Abstract algorithmics will be concerned with the depiction and analysis of complex algorithmic processes, independently of the way in which the logical objects to thich they refer are to be mapped into a computer. Much of the present work is intended as a first illustration of what abstract algorithmics is ultimately to be. Concrete algorithmics, on the other hand, has the following as its problem: Given a family of abstract objects and processes that are to affect them, how can these objects best be mapped into tabular form and the associated processes actually carried out? Note that by isolating and first solving some of the problems of abstract algorithmics, we may expect to be able to discuss concrete algorithmic problems in a more satisfactory manne

than has hitherto been typical. Before one knows what one wants to do in a complex situation, one is really not in a very good position to study the ways in which one might do it. Conversely, once an abstract algorithm is put forth, one is generally able to envisage a much wider range of concrete approaches to its optimization than would otherwise be possible. From the point of view of the present work, prior systematic attempts at the depiction of algorithms have generally failed to separate the abstract from the concrete parts of the algorithmic questions that they study but have mixed the abstract with the concrete, rather to the disadvantage of both. It may also be mentioned that it is abstract rather than concrete algorithmics that stand closest to a third principal branch of computer science, that which plays so large a role in Donald Knuth's magnificent series of books, namely the *formal performance analysis of algorithms*.

A second benefit is this: The succinctness and descriptive power of a mathematicized programming lanugage will enable us to depict complex processes in their totality, in decisive detail, and in a form free of abstractly irrelevant specifics. Since our view of complex abstract algorithms will be total and detailed rather than fragmentary and vague, we will have a better chance to consider the form and effect of variations in our algorithms and the possible generalizations of them. Bringing the abstract kernel of a process out from behind the veil of abstract irrelevancies that normally obscure it, we will make this kernel more communicable and hence more capable of systematic rational discussion than would otherwise be the case. The elimination of irrelevant specifics from formally stated algorithms has still another substantial benefit. Specificity is a major source of incompatibility between algorithms, and abstractness will therefore enhance compatibility. Consider, for example, the problem of piecing together a compiler out of its customary principal components: a parser, optimizer, code generator, etc. We find, typically for a whole class of similar situations, that across each of the interfaces between principal modules some collection, generally rather small, of structured data items must be passed. From the

abstract point of view, these will be unproblematical enough: They may be trees, graphs, a few mappings defined thereupon, and so forth.

If one module will naturally produce trees or graphs, and another can conveniently accept such as an input, no serious problem of compatibility is to be expected at the abstract level, and separately written modules can readily be fitted into a totality. The situation immediately becomes different when abstract data structures are mapped into concrete tabular form. To do so is to define a host of pointer and indexing mechanisms, supplementary variables, overflow conventions and flags, special abbreviations relating to particular data subcases, field sizes, punctuations, and so forth. These definitions, precisely because from the abstract point of view they are largely arbitrary, will never be cast in precisely the same way for two separate program modules except by careful preplanning, and, of course, any deviation from perfect agreement may require data restructurings so complex and touchy as to be prohibitive. In this we have the case that generally makes it impossible to design useful program library components that will either accept complex data structures as input or provide such structures as output, unless, as is the case with the SETL specification language, a language of sufficient generality provides a fixed framework of conventions. For this reason, intermodule data interfaces often become the foci of major difficulties during the design of large programming systems, wherein months are often consumed in negotiations concerning the detailed layout of data structures to be passed between principal modules. It is usually found in such situations that all the technical groups involved have elaborated their design ideas not abstractly but in terms of certain implicit assumptions concerning data layout, assumptions that, understandably enough, they become loath to give up. Moreover, no powerful algorithmic communication technique permitting one group to gain an understanding of the processes that the others are going to apply is available, so that the trade-off issues involved in the choice of data layouts tend to remain obscure on both sides. Naturally, it is then hard to come to intelligent technical compromises.

The systematic preelaboration of algorithmic strategy using a powerful specification language should cast a welcome light on this dark and perilous corner.

Given that the elimination of irrelevant specifics will restrict the tendency toward incompatibility of program modules in a significant way, we may expect to be able to produce fairly complex standard formal algorithms adaptable for use in a variety of situations. Thus it should become possible to put larger items into the cabinet of prefabricated programs than have hitherto filled it. In the present work effort will be devoted to doing just this.

Yet a third anticipated benefit that we expect the use of a mathematicized specification language to provide is this. Our method allows the abstract specification of an algorithmic process to go forward to completion before any of the concrete table-and-code design issues connected with it have to be faced. When our language is implemented, it will even be possible to execute the abstract programs, and to verify their correctness experimentally. During this process a much smaller mass of program text will be involved than is now the case; it will be much more feasiable than it now is to experiment with significant variations in approach during the development of an algorithm. Next, having a debugged algorithm in hand, one will be able to survey it to get a detailed picture of all the data structures that it involves, of all their parts, and of all the processes that must effect these parts. It should then be easier than it is now to come to sophisticated final decisions concerning the table structures, data management strategies, and code techniques to be used in a highly efficient version of the same algorithm. In current practice, both classes of issues, abstract design and concrete layout, must be faced at once, and generally in a situation of confusion in which a programmer, already forced to cope with all the complexity that he can juggle, may be unwilling even to contemplate promising optimizations if they threaten to add to the mass of material that he must sustain. Moreover, in typical current situations it is quite hard to maintain design balance, with the consequence that certain parts of the system

may be overdesigned, while others, equally or more crucial, may be sorely neglected and their insufficiency discovered only when it has become impossible, or at least extremely expensive, to do anything about it. The algorithmic language that we propose should, in short, allow the full complex of program design issues to be approache in orderly stages and allow minor matters to be classified as such, whereby design attention can be concentrated to ferret out highly effective solutions to key problems.

Note also in this connection that it is only by providing a formal language allowing one to describe the abstract structure of algorithms without implying any commitment to some particular concrete realization that a systematic formal attack on the problem of concrete algorithmics becomes possible. If the initial text of an algorithm is infected with some particular view concerning certain aspects of the data structures to be used, it will not be possible to 'declare away' these implicit assumptions, and one's ability to move subsequently to an efficient realization of an algorithm may be lost. Present practice however makes it very hard to retain flexibility in regard to data strategy. In the absence of languages allowing the structure of an algorithm to be described abstractly, data structure design is often the first step taken by a programmer. Presently, a sketch of fields and tables is often the first thing written down as a programmer attempts to turn the exterior specifications for a program into some early idea concerning the program's internal workings. This step, fraught with profound consequences for all that will subsequently happen, is in current practice taken immediately, even before any consistent algorithm is available, simply because at present some amount of data structure design must be carried through if a vocabulary for the detailed description of an algorithm is to be established. An abstract programming language will as a fundamental benefit not only make it possible to postpone discussion of concrete algorithmic issues until the abstract part of an algorithm design has been brought under control, but will also make it possible to address the problem of concrete algorithmic design declaratively.

Most of the benefits to which we have until now alluded come rom the use of a suitably powerful specification language, independently of whether this language is implemented. When the mathematicized programming language we project is available for running, however, additional advantages will accrue. A language of this kind cannot but be a most appropriate tool for those situations, especially charactersitic of university programming, in which experimental algorithms are developed to be run a few times and then improved or discarded after certain aspects of their behavior are observed. A tool of this kind will also be useful when an elaborate program needs to be built to run just once, or when a complex program whose sole task is to prepare tables for some other program must be produced, or when meta-compilers or other large programs of infrequent use must be prepared, etc. The mathematician desiring to experiment with combinatorial situations but unwilling to make a very heavy investment in programming will find a language of the type orjected most welcome. The computer scientist will find that it allows him to realize more elaborate algorithms than would otherwise be in reach. Such attractions have made APL increasingly popular; I consider that the partly set-theoretic nature of that ingeniously devised array language lies at the root of this phenomenon.

In the development of large programming systems within an industrial setting, a technique allowing the rapid and inexpensive development of functioning, even if inefficient, versions of complex programs must also be of decided advantage for several reasons. Presently, large-program development suffers badly from the fact that little or nothing begins to function visibly until a huge whole has been brought far along. At this point, vast sums may have been spent and time irrecoverably invested. It is then generally the case that what is done is done and that a project must either bull through along a fixed course, whatever its internal or external deficiencies, or die. Simply by shortening the perilously long feedback loops that characterize present development techniques, an executable specification language would prove of reat advantage.

It may also be remarked that during the development of a large system substantial expenses are occasioned by the fact that in such projects it is often necessary to write masses of scaffold-code against which developing systems components can be tested. Intending that our mathematicized specification language should obviate this, we have been at pains to specify an interface linking the specification language to a conventional field-manipulation language of the kind that would normally be used for systems programming. In tandem with such a "lower level" language, our specification language can act as a test case generator.

The full development of this idea leads to what might be called a two-stage development technique. The first of the two programming stages consists of the development and debugging of a complete systems algorithm, written in the abstract language, and the annotation of this algorithm with all those remarks concerning intended concrete techniques and data management strategy necessary to define the detailed program that is to be developed during the second stage. This first programming stage will also involve measurement and user testing, wherein the abstract algorithm serves as a kind of detailed simulator of the efficient program that it foreshadows and wherein it may be modified as necessary. An optimizer using the data strategy declarations which we envisage might in many cases produce a running code quite acceptable in efficiency, without any reprogramming being necessary. If in a true 'production situation' a more highly efficient version of the same code were required, a reprogramming phase would be required. During the second stage of programming, all the parts of an abstract code are progressively replaced with logically equivalent but much more efficient passages of concrete code, which are hammered out against the abstract algorithm.

All plans involving execution of abstract specification language must eventually hope to demonstrate that, even though the efficiency losses that generalized and standard data representations must occasion will be large, they need not be catastrophic. Loss factors of 10, or even of 100, can be borne; loss factors of 1000 (which we do not at all anticipate) would be disastrous. Technology has, after all,increased memory capacities

by a factor of 100, and speeds by even larger factors, over the ast dozen years, and promises to continue making similarly spectacular gains. Would it not for many purposes be clearly worthwhile to go back a generation in machines if by doing so we could increase by a large factor our ability to program?

As a final benefit, we expect the availability of a mathematicized algorithm specification language to broaden the frontier of contact between programming and mathematics. It should at any rate serve to emphasize to the mathematician that programming need not be a mass of petty detail only, that in fact it is concerned, in a way only slightly unfamiliar, with some of the issues that he is accustomed to confronting, that interesting inductive proofs can in fact be regarded as recursive algorithms, and so forth.

While the mathematician will presumably find a mathematically oriented language like that we propose more familiar, and hence more accessible, than customary programming languages might be, the programmer coming to it with a conventional background will find it necessary to change certain of his central habits, and this may at first be rather disconcerting. Conventionally, the mental process of program elaboration that eventually results in a finished program design or program begins not only with half-formulated procedure kernels but as much as anything else with some idea of the data structures that are to support the procedures to be employed. Often, enough the first part of a total design that appears on paper is an initial elaboration of these data structures, their fields, and the separate significances of these fields. This data depiction is conventionally used as an anvil against which all the detailed processes that eventually will form part of a complete package are shaped. From the present point of view, all this -- ingrained habit of the most skillful programmers though it is -- is defective, since it indiscrimintately confounds the abstract essentials of a process to be described with a host of matters of quite different character. The procedure we suggest is different. Bypassing very much of this customary matter, or at the very least making it a postscript to rather than the start of our specification process, e deal not with *tables*, *fields*, and *pointers*, but directly with

those *logical associations, correspondences,* and *sets* that conventional tabular data structures ultimately and indirectly represent. The sudden loss of burden that so radical a simplification implies may at first be somewhat disorienting, and the new medium may at first seem too rarefied to breathe. Nevertheless, the necessary new habits of thought are in fact readily acquired and, once mastered, can lead to a substantial improvement in one's ability to design eminently practical algorithms. But the necessary design steps will be taken in quite a different order.

Before coming to the end of this preface a few last generalities may appropriately be put to paper. What are programming languages? We would like to suggest the following outlines of an answer to this question, evidently fateful for any effort at language design. Programming lanuages are notational systems devised to facilitate the description of abstract objects whose basic elements are sets, mappings, and processes. Associted with these objects will be a well defined rule for evaluating them; perhaps, since the objects may contain processes as subparts, it would be better to say, for interpreting them. From this point of view, procedural programming languages of the ordinary serial kind may be regarded as a mechanism for the description of a set of basic transformation blocks, with each of which is associated a family of possible successors. Each block must also be furnished with a "terminating conditional transfer," which can be used during interpretation of the program to select one potential successor block as the actual "point of transfer." Object describing languages will depart strongly from this familiar kind of location counter control, however. In simulation languages, for example, the basic principle of control is quite different: The subprocesses of a simulation naturally form an unordered set, each of which is furnished with an invocation condition. The simulation interpreter executes, in any order, all processes whose invocation condition is satisfied as long as any remain to be executed; when none remain, an underlying time parameter is advanced by the interpreter, and the next cycle of simulation begins.

If we bear in mind this broad range of possibilities, we can sharpen our response to the question posed above as follows: The "front" or "syntactic" part of a language system must provide methods by means of which very general abstract objects (graph-like rather than tree-like, i.e., admitting remote rather than purely local connections) can be described conveniently. This "front end" should be variable enough so that the descriptive notation to be used can be tailored to the requirements of any particular field, permitting the objects of most common concern in this field to be described in a succinct and heuristically comfortable manner. Powerful mechanisms for describing the diagnostic or verification tests to be applied to text during its syntactic analysis should also form part of this language-system front end. The "back" or interpreter part of a language system should incorporate abstract structures that are general enough so that all the structured objects that may be of concern in a particular situation can conveniently be mapped upon these structures. Now, the use of general set theory should certainly satisfy this latter requirement, as long as the actual use of theorem-proving methods is not at issue. Set theory could only fail to be adequate if some other entities than sets were directly accessible to mathematical intuition and could therefore be used as a fundamental starting point independent of set theory, which is not the case. Thus, if a suitably flexible syntactic front end can be attached to the set theoretic language with which we shall be working, we will have a system covering a good part of all that is likely to be found along that road which completely bypasses considerations of efficiency. This will in fact be attempted. Of course, this will still leave room for semi-general languages which compromise artfully with full generality in order to reach higher efficiencies than would otherwise be attainable.

Item 5. INTRODUCTORY DISCUSSION OF SETL.

In the present work we will propose a new programming language, designated as SETL, whose essential features are taken from the mathematical theory of sets. SETL will have a precisely defined formal syntax as well as a semantic interpretation to be described in detail; thus it will permit one to write programs for compilation and execution. It may be remarked in favor of SETL that the mathematical experience of the past half-century, and especially that gathered by mathematical logicians pursuing foundational studies, reveals the theory of sets to incorporate a very powerful language in terms of which the whole structure of mathematics can rapidly be built up from elementary foundations. By applying SETL to the specification of a number of fairly complex algorithms taken from various parts of compiler theory, we shall see that it inherits these same advantages from the general set theory upon which it is modeled. It may also be noted that, perhaps partly because of its classical familiarity, the mathematical set-notion provides a comfortable framework for thought, that is, one requiring the imposition of relatively few artificial constructions upon the basic skeleton of an analysis. We shall see that SETL inherits this advantage also, so that it will allow us to describe algorithms precisely but with relatively few of those superimposed conventions which make programs artificial, lengthy, and hard to read.

Having general finite sets as its fundamental objects, SETL will be a language of very high level. Generally speaking, we regard the level of a language as being high to the extent that it succeeds in getting away from the requirement of strict locality of operation which adheres to an elementary automaton. That is, a high level language is one which incorporates complex structured data objects and global operations upon them. Such a language can free its user from the onerous task, artificial from the abstract structural point of view, of specifying the detailed internal tables which are to represent the structured objects of his concern and of reducing to purely local functions the global transformations which affect these objects. The programmer is thereby freed to write of abstract problem-related entities and their

interactions in a familiar and analytically natural manner.

As we have noted, a price must be paid for these very great advantages. A high level language, which reduces to a minimum the amount which a programmer must write to specify an algorithm in executable form, is apt to become committed to the invariable use of certain standard tabular forms for the representation of the entities with which it is concerned, and to the use of certain standard procedures for their manipulation. It will generally use these tables and procedures even in cases in which the nature of a particular process to be programmed allows the use of much more efficient data representations and manipulations. Thus, the use of languages of very high level will lead in many cases to the generation of very inefficient programs. To the extent to which we are unable to capture the optimizing inventiveness of a skilled programmer by an optimization algorithm, this difficulty will persist quite generally. Set theory, which in principle regards a function like cos and the set of all pairs x, cos(x) which it generates as being equivalent, certainly tends to the use of dictions implying highly inefficient algorithms, and whereas we will find ways to avoid the worst of these in SETL, and will discuss various ways in which SETL programs can be optimized, it will still be true that SETL will pay a substantial price in efficiency for its logical power.

Nevertheless, it is our feeling that this substantial objection is not catastrophic, i.e., that languages of the type of SETL can be quite uesful in a variety of significant situations. SETL will, in the first place, be useful as a specification language, i.e., as a language in which algorithmic processes can be formally and precisely defined by a text whose *syntactic* correctness and completeness may be verified computationally. The value in the definition of highly complex objects of the use of formal text has been emphasized by researchers at the IBM Vienna Laboratory (cf. Lucas, Lauer, and Stigleitner [1]), who have developed a logical metalanguage called ULD incorporating various powerful set theoretic features and have applied it to give a formal definition of the PL/1 programming language; we will make various

comparisons between SETL and the Vienna ULD definition language below. It may be remarked that the use of formal text for the definition of algorithms is also a step toward the verification of algorithm correctness and equivalence by formal proof methods. Unfortunately, such a step does not bring us very far toward this rather distant goal, as algorithms of the sophistication

There are other uses than formal definition to which a powerful but rather inefficient programming language like SETL may be put. In certain situations, fairly complex algorithms must be programmed in order to run just once; this is the case, for example, whenever a new languages is being "bootstrapped" into existence. In other cases, as for example in experimenting with complex algorithms, in measuring the performance of such algorithms, or in the simulation of complex systems, it may be appropriate to program elaborate

procedures to be executed just a few times. In such cases, lgorithm inefficiency may be of minor importance.

Still another significant advantage of the availability of a language of very great expressive power can come from the fact that it makes a formal data strategy declaration language possible. In such an approach, algorithms to be progressively improved in efficiency would first be written out in SETL, no data strategy declarations initially being supplied. Once the verified algorithm were available, one would begin to improve its efficiency. Examining the algorithm, and noting the sets which played an important role in it as well as the operations to which these sets were subject, one would elaborate a specialized *data description* which could lead to an efficient program for the same algorithm. Note in this connection that sets of certain kinds appearing frequently in programming situations permit various specially efficient representations. Thus, for example, a subset B of an explicitly represented set A can be represented by a collection of one-bit flags logically attached to the elements of A, and either housed with these elements or segregated into a bit table; a set of ordered pairs may be represented by a two-dimensional array, possibly of bits; an interval of inegers may be represented implicitly by a numerical range; etc. Many such representations are known, and constitute much of the normal stuff of programming. To extend such a catalog of techniques, we may note that sets can be represented by arrays, either sorted or unsorted; by lists structured using pointers; by entries in hash tables; or by data structures combining all of these representational methods, as for example structures in which a first few set elements are represented in one way while the remaining elements of the same set are represented in another. Sets of ordered pairs may be represented by attaching all those second components B which occur with a given first component A as a list; alternatively, all the elements of this list may follow A in a packed array, permitting the representation of a set of ordered pairs in a highly compressed form. Special bit-table techniques are available for the representation of functions defined on thin subsets of a known set, etc.

Special systems of grouping, advantageous when some part of the total representation of a set is to be stored on an external storage device, are also known. Later in the present manuscript, a data strategy description language, DDL, whose semantics gives systematic embodiment to the observations made above, will be proposed. The use of this declaratory language will be illustrated by giving annotated forms of various algorithms.

Since one has generally lacked convenient means for the formal description of abstract algorithms, the varied techniques described in the above paragraph, which may collectively be said to constitute the *concrete* part of the theory of algorithms, have more often than not been discovered in intimate connection with that rather different material which we prefer to think of as *abstract algorithmics*. SETL will allow us to deviate from this tradition, and to separate, into two distinct stages, the problem of designing and precisely describing the abstract structure of an algorithm on the one hand, and the different problem of mapping this algorithm efficiently onto a given machine by choice of data structures and procedures coded in a language of lower level on the other. Such separation will hopefully allow us to be more accurate in our treatment of both problems than we could normally be if, as is now generally the case, we were constrained to treat them together. It may in particular be noted that a given abstract algorithm can have many plausible concrete images, some of which might be missed if both the abstract and concrete form of an algorithm have to be designed together. We may also hope that the availability of a tool like SETL will enable the accumulation in useful form of "prefabricated" abstract programs. Because abstract programs are free of much of the specific detail which associates itself with algorithms in more concrete form, prefabricated abstract algorithms of this kind might serve as permanently useful blueprints for the development of efficient concrete programs. At any rate, we can regard the development and debugging of an abstract algorithm, and its annotation using a formal data strategy declaration language, as completing the first stage of a two-stage programming process. In an industrial setting, this

might mark the termination of an initial design phase and the passage of responsibility from a design group to a group of final implementors who were to develop a high efficiency production version of the same code.

The utility of SETL during the elaboration of this efficient final program can be enhanced by linking SETL explicitly to a more conventional programming language of lower level, which in this introductory discussion we shall designate as the lower language or LL. LL can be any small "systems programming" oriented language, that is, any language oriented toward the definition of structured tables and their efficient manipulation. In the total language configuration envisaged here SETL and LL are both incorporated in a manner allowing them to be intermixed. In particular, once an efficient tabular representation for some structured set S occurring in a SETL program has been decided upon, it will be easy, using an appropriate mixture of SETL and LL phases, to describe procedures both for converting S to a tabular representation T and for converting a table having the structure T back to a set structured as S. Such conversion being possible, one may then progressively replace sections of SETL code by expanded versions of the same processes written in LL, converting between tabular and set-theoretic representations for all necessary entities at the points of transition between SETL and LL code. Holding to such a procedure will aid in the orderly elaboration of a final program, and will also provide some assurance that the program as finally developed corresponds to the SETL specification as initially set down. As final LL code is developed, initial SETL statements will progressively be relegated to the status of comments. During this process, those portions of an original SETL code still being executed will serve as "scaffolding" for the developing LL code, providing input data sets and convenient output and checkpoint facilities for testing. Note, however, that as long as a program remains "hybrid", that is, contains both SETL and LL, it will be inefficient, as the costs of passage of the SETL-LL interface that we shall describe are high.

SETL, as it will be presented in most of what follows, will appear as a full-fledged "user language". That is, we will assign it a specific and fairly elaborate syntax, intended for direct use rather than as a base requiring a good deal of extension before use. Here we may usefully distinguish between "host" and "user" languages. A *host language* is a language providing a full set of semantic facilities, but with a syntax deliberately kept simple. Such languages are intended not for direct use, but rather as a basis and target for language extension. By keeping their syntax simple and modular, one confines the mass of irregularities which an attempted extension must digest. In designing a *user language*, on the other hand, one incorporates a fairly elaborate collection of syntactic facilities, hoping that these will be directly useful in a wide range of applications. SETL as currently specified is a user language; it is however worth trying to envisage the sort of host language which could underlie it, as this will clarify various basic semantic issues arising in programming language design.

At the most basic level, a procedural programming language merely provides a framework which allows the storage and retrieval of some family of data objects and the sequenced application to these objects of some sufficiently general set of primitive operations. Concerning this, the following remarks can be made.

1. The semantic framework of a language should allow an arbitrary combination of primitives to be disguised as a primitive and to be invoked in the same way as a primitive. This is the essential point of the system of "calling conventions" which is always part of the semantic core of a programming language. One will wish not only to provide subroutines capable of manipulating and modifying their arguments, but also to provide from the start for the recursive use of subroutines, as recursion is a technique of established power related directly to a language's basic inter-routine linkage conventions.

2. Computer hardware characteristically looks at computations through a 'peephole'. In each cycle of its action hardware can manipulate only those several hundred bits which are contained in some limited set of active registers. Moreover, internal limitations on available data-flow paths will slightly (though not strongly) constrain the transformations possible on a single cycle. To the extent that its primitives directly reflect hardware constraints, a programming language is of low level; to the extent that it provides compound data objects upon which highly 'global' primitives act, a language is of high level.

3. If compound data objects are to be freely usable in a programming language, and if processes creating such objects are to be usable in the same way as hardware primitives would be, then at the implementation level the language must incorporate some automatically acting space allocation scheme. In this sense, we regard the type of "garbage collector" first developed in connection with the LISP language as an invention fundamental for programming.

4. The semantic extensibility of a language will depend to a large extent on the ease with which different abstract compound objects of varied structure can be represented in terms of the specific semantic objects which a language provides. SETL aims to gain advantage from the fact that the objects of most fields of mathematical discourse can be represented rather easily using sets and mappings.

5. The primitive operations most advantageously incorporated into a language are those which combine smoothly into broadly applicable families of transformations, especially if they are simple, heuristically comfortable actions which are nevertheless challenging to program. Some at least of the set-theoretic operations provided by SETL have these advantages; note in particular that efficient implementation of set theoretic operations will probably imply the use of hashing techniques, and that the heuristically innocent notion of set equality is implemented by a fairly complex recursive process.

6. A programming language built upon a family of primitives can be optimized if enough information concerning the inputs and effect

of each primitive is made available to a sufficiently powerful optimizer. Optimization will normally require extensive processing of program text, and one's attitude toward optimization will therefore have important impact on a language design. In general, a decision to optimize extensively will imply a relatively static approach to certain design issues; if optimization, especially global optimization, is abandoned, a considerably more dynamic, incremental language can be provided. This observation bears upon the question of whether some appropriate intermediate-text form of its own code is to be a data type available within a language. If this is done, then highly flexible forms of dynamic compilation become available. In the opposite case, compilation (with the optimization it implies) is a more serious step, and in the place of truly dynamic compilation a language system will probably provide a facility for input text editing plus total recompilation. Dynamic compilation is certainly more than a luxury; on the other hand reasonably flexible and quite useful systems can exist without it. Of course, it is possible to provide both modes of operation, one for the earliest stages of program development, the other to allow more substantial runs of more fully developed programs.

7. An optimizer will normally require substantial information concerning all the primitives which it can encounter. This information will be used in complex ways during the analyses which constitute the optimization process. For this reason, it will not be easy to allow extension of the semantic primitives of a language which is to be elaborately optimized. On the other hand, extensibility of primitives of a high level language is quite desirable, as this will allow levels of efficiency to be reached which can probably not be attained in any other way. Note, for example, that SETL provides no SNOBOL-like string-to-pattern matching primitive, a primitive which would be highly desirable if SETL or an extension of it were to be used for extensive string-processing. Similarly, it might be highly desirable to provide, let us say, a set of matrix manipulation primitives in an extension of SETL which was to be used to support large scientific calculations. To keep open

the possibility of adding such primitives is certainly not a goal hich it is easy to accommodate if, as in SETL , careful optimization is to be undertaken. Nevertheless, it may be hoped that careful organization of an optimizer will lend a rational structure to the information concerning primitives which an optimizer must use. To the extent that the hope is borne out, it may be possible to allow new primitives to be installed within an optimized high level language. This will of course require that certain interface conventions be observed carefully, and that information of significance to the optimizer be supplied in prescribed form whenever a new primitive is established.

8. The use of high level dictions should, at least to a certain extent, make a language more optimizable. It is probably easier to supply an optimizer with information concerning important special cases of high level operations than to enable it to detect "gestalts' once these have been expanded into detailed code sequences. Appropriate combination of high level procedural code with declaratory hints to an optimizer may very possibly make possible the production of rather efficient object code from concise and straightforward source text. For this to succeed, however, we will have to learn to express essential aspects of a programmer's overall optimization strategy in a suitably devised formal declaratory style.

9. A facility for the description and control of processes proceeding in parallel is vital for languages intended for certain important areas of application. Such a facility can be provided either in a broad form suitable for use in connection with true multiprocessor-multitasking environments, or in a narrower form sufficient to allow response to semi-autonomous external activities generating 'input data files', 'clock signals', and 'interrupts'. At least the latter facility is necessary for a language which is to be used to describe operating systems. While SETL does not now include any features of the type just described, it is certainly interesting to attempt to extend SETL to provide such features. Errors and error recovery, as well as memory hierarchy management, raise additional ignificant operating-system releated programming issues with which a language intended for the comprehensive description of such

systems must come to grips.

10. The provision of a structure which can supply good run-time diagnostics is an issue which ought to be faced during the basic semantic design of a language. Diagnostics of this sort will be issued when the routines interpreting the primitives of a language detect malformed arguments or other illegal circumstances. In such cases, 'trace-back' data allowing the situation surrounding an error to be described in user-interpretable terms must be available. One will wish to be able to report on

(a) the chain of subroutine calls made up to the error event;

(b) the source-code statement, and the particular operation in it being executed at the time of the error event;

(c) the values of all variables immediately prior to the error event, and the identity of those variables discovered to be illegal in format.

The importance of providing all this information in an 'external' form tying it to the source text with which a user is directly concerned, rather than in a difficult-to-interpret internal form, deserves to be stressed. Note that many of the necessary 'external-to-internal' connections (such as the association of external names with variables) can be set up at compile time by an adequately conceived translator. Nevertheless, the diagnostic 'hooks and eyes' needed at the basic execution level deserve careful design.

The issues discussed above are strongly semantic in flavor, in that they arise during the design of the base-level interpreter routines and target-code conventions which directly allow the operations of a language to be carried out. Beyond these issues arise others, still fundamental, but of a more nearly syntactic character.

We may regard these latter issues as belonging to the design of those processes which take one from some internal parse-tree form of a host language to the target code which is directly interpreted. Here, the following main points may be noted.

1. In designing a full host language system, one will have to decide whether the system is to include a farily complete 'front end', or whether only host language mechanisms will be provided.

If the former path is taken, one will strive to invent 'syntax xtension' tools allowing the external form of a language to be varied within wide limits. If only host language mechanisms are provided within the core system, one will intend to allow any one of a wide variety of parsers to be used to define external language syntax. The first course can provide quite a range of users with languages reasonably well tailored to their requirements, which can be made available without any very great effort on their part. Several arguments can be given in favor of the latter course. Parsing is the best understood, the most manageable, of all the elements of a language system. Diagnostic generation is an important part of parsing, and a specially designed parser can generally give much more adequate diagnostics than are available via a less flexible syntax extension scheme. In particular, the use of a syntax extension scheme may make it difficult to avoid t e generation of diagnostics at the host language level, which however may involve the user in dictions and circumstances that he would prefer to know nothing of. A predefined syntax extension scheme may not readily allow the use of source text forms requiring elaborate, unexpected pretransformations, as for example forms in which initially segregated fragments of code must be merged to produce required host language forms. Especially if this merging involves elaborate consistency checks, or is guided by specialized declarations peculiar to a given user language, attempts to use a pre-defined extension scheme lead to difficulties.

2. Even a host language will generally provide more than the minimally necessary operations, argument, and transfer patterns required to sustain interpretation (a language providing only this much would in effect be an assembly language for an abstract machine). Indeed, since some basic elements of syntactic 'icing' are so easily provided, and so apt to be useful in connection with almost any ultimate external syntax, one will generally wish to provide at least this much syntax as part of a host language. The list of features which one will probably prefer to include is airly short. Expressions with embedded function calls are a

syntactic form used in almost every programming language. They derive their special utility from the fact that the output of one operation is quite likely to be an input to the very next operation performed; when this is true, use of expression syntax allows one to elide the 'temporary variable' names which would otherwise have to be used, yielding condensed dictional forms. The 'on-the-fly' assignment (within an expression) pioneered by APL can be regarded as generalizing this advantage; it exploits the fact that one will often use the value of a subexpression twice in rapid succession, often within the confines of a single compound expression. Certain dictions related to the control of program flow have equally general appearance, and deserve equally to be provided even by a host language. The *if ... then ...* form popularized by ALGOL exploits the fact that binary tree-like branching is the commonest form of 'forward' conditional transfer. By providing this diction at the host language level, one eliminates the need to generate many of the explicit transfer labels which would otherwise be necessary. The commonest form of control structure involving backward branching is the 'while' loop, which is another form which it is desirable to include even in a host language. One will wish a collection of statements to be usable in any position in which a single statement can be used; for this reason, it is desirable for a host language to include some statement grouping scheme. Finally, one will wish to be able to use any code passage returning some single data object as part of an expression; a facility allowing this is also appropriate for a host language.

3. Name protection, embodied in a suitably powerful and general namescoping scheme, will appropriately be included in the host language level of an overall language system. We regard a namescoping system as a set of conventions which assign a unique 'resolved name' or 'semantic item' x to each 'source name' y appearing in a mass of text. The particular x to be assigned to each occurrence of y depends on the location of x within what will ordinarily be a nested, tree-like family of *scopes*.

The purpose of a namescoping system is of course to balance the conflicting pressures toward globality and protection of names.

Unrestrictedly global use of names is unacceptable, since it creates a situation of 'name crowding' in which names once used become, in effect, reserved words for other program sections. Hard-to-diagnose 'name overlap' bugs will abound in such situations. 'Globalization' of any subcategory of names can recreate this problem; for example, in large families of subroutines it may become difficult to avoid conflicts between subroutine names. In sufficiently large program packages, it will be desirable to give even major scope names a degree of protection.

On the other hand, a system in which names tend very strongly to be local unless explicitly declared global can tend to force one to incorporate large amounts of repetitive declaratory boiler-plate into almost every protected bottom-level namescope or subroutine. Particularly in a language like SETL, which aims at the compressed and natural statement of algorithms, this burden would be irritating.

What one therefore requires is a system capable of dividing a potentially very large collection of programs into a rationally organized system of 'sublibraries', between which coherent cross-referencing is possible in a manner not requiring clumsy or elaborate locutions.

The design of such a system is by no means trivial, especially since the problems which namescoping addresses emerge full-blown only in the development of very large systems of programs. Note also that a namescoping scheme to be used in connection with an extensible host language ought to be general enough to support a variety of user-level namescoping conventions. The stereotyped subsidiary text necessary to get such a variety of surface effects will of course be supplied by the specialized 'front ends' defining the different user languages supported by a given host language. However, before any of these issues can be faced with confidence, more experience is required.

Having said what we can concerning the basic semantic and semi-semantic issues arising in language design, we now turn to a discussion of some important syntactic issues. Any language syntax will 'fill' a given space of syntactic possibilities to a given

level of completeness. Of course, one will never wish to assign a meaning to every possible string of tokens; to do so would completely destroy all possibility of detecting error during translation. On the other hand, it can be advantageous to allow a language to fill the syntactic space available to it rather completely; say, for purposes of discussion, to the 50% level. This will tend to make many very compact dictions available; a possibility especially attractive if an interactive keyboard language is being designed. To attain this level of syntactic packing, one will assign meanings to operator-operand combinations not ordinarily used, and reduce the number and length of keywords appearing in a language. In particular, monadic significance will be assigned to ordinarily dyadic operators, a semantic interpretation will be assigned to sequences of variable names following in succession with no intervening operators, and elision of keywords willbe allowed whenever possible. A precedence structure favoring the use of infix operators over parentheses may also be found desirable. All this leads to a very compact language, in which helpful syntactic diagnostics can nevertheless be generated. Syntactic packing to the degree indicated may on the other hand lead to source text forms which, lacking helpful redundancy, become somewhat difficult to read. For this reason, one may prefer in designing a language intended for extensive use in cooperative programming efforts to make use of a higher degree of redundancy. In such case, the syntactic structure chosen ought to promote good programming habits, allowing and even inducing its user to group passages of text in a manner which makes clear the logic of the process which this text describes. Moreover, as a return for the redundant modes of expression imposed upon him, the user can gain the use of a subtler and more complete set of compile-time consistency checks.

It is desirable to include a fairly powerful macro-preprocessor in the front end of a language system. This will allow the particular syntax provided by language to be "perturbed" in ways a user is bound to find convenient. In particular, local abbreviations can be introduced, minor special dictions set up, etc.

Macros with parameters, nested and conditional macro-calls, macro iteration, and a certain amount of compile time calculation power are all desirable. More elaborate built-in string transformation schemes, which involve the parsing of an input string into a tree form which is then transformed into an output string, can be used to give a higher degree of syntactic variability to a language. Of course, the more far-reaching a transformational scheme of this sort, the more delicate is its correct use apt to become.

In the ordinary course of a syntactic design, the most desirable syntactic constructions will be used at once; if they are not reusable, less than optimal dictional forms will have to be employed subsequently. Note, for example, that depending on context one might want a*b to denote the product of numbers, the dot-product of vectors, the product of matrices of of group elements, the intersection or cartesian product of sets, as well as any one of a great number of vaguely 'product like' constructions occurring in other application areas. We see the solution of the dilemma implicit here as lying in the use of a mechanism important in natural language usage. Namely, the interpretation of syntax must depend on context; specifically, the manner in which an operator applies to an object (or collection of objects) should depend on the object's nature. Thus we find it desirable for a linguistic system to incorporate a formal mechanism allowing the definition of indefinitely many different 'object kinds', which can be used to control the manner in which statements of a fixed external appearance are interpreted. Such an approach has in fact been tried in a number of languages, sometimes on a dynamic (run-time) basis, sometimes on a static (compile-time) basis, occasionally in a manner having both dynamic and static features. In a later section, a static system of object kinds will be proposed for SETL. A static rather than a more flexible dynamic approach may well be adequate, and does not imply any loss of efficiency. The system proposed will probably also be useful in debugging.

Item 7. SETL IMPLEMENTATION AND OPTIMIZATION. A FIRST LOOK AT SETL COMPILATION: TARGET CODE STYLE.
(Revision of SETL Newsletter 53: H. S. Warren Jr.)

The following pages discuss various questions of target code style basic to the compilation of SETL. In such a discussion we must of course intend some lower level language (which might possibly be the machine language of a particular computer) as the target language of compilation. In the present version of the SETL compiler, the target language adopted is an NYU-developed "systems programming" language intended for the writing of production compilers, operating systems, etc., and called LITTLE.

LITTLE is roughly a subset of FORTRAN, at about the level of BASIC, augmented by field extractors. A main property of LITTLE is that programs coded in it are highly portable. Portability is achieved by allowing only two data types: fixed length bit strings, and floating point numbers of fixed (but machine dependent) length and form. Fixed length bit strings are treated as unsigned numbers by the LITTLE arithmetic operators.

LITTLE is designed to be highly optimizable in a global sense. It includes no pointer or label variables and no recursion mechanism. It allows only single subscripts on array references. A macro processor of simple string replacement type is provided; macros may contain parameters. Field extractors are provided. For example, X = .F.3,2,A extracts the 2-bit subfield which begins with the third bit of the quantity A. The field extractor may also be used in a "sinister" context. For example,

```
.F.3,2,A = 0
```

turns off the bits in a two-bit subfield. The macro-processor provided as part of LITTLE allows us to refer to subfields by name. Thus we may write the above field-operations as

```
X = TWOBITS A        and        TWOBITS A = 0
```

respectively.

Dynamic Storage

The SETL implementation employs a compacting garbage collector. Available storage is divided into two areas, as depicted on the following page. A run time stack is maintained at one end, an allocatable "heap" at the other end. The stack is used in the usual way to support procedure calls and returns. The compiler maps every variable in a program into a stack location of the form (base + offset), with *offset* a constant. The stack holds *root words* describing each variable's value (these root words are described below).

The heap is used to fulfill requests for storage. Sets, long character strings, etc., are all stored within the heap. Requests for heap storage are satisfied in a simple linear fashion. If the normal response to a request for either heap or stack space would cause the stack and heap to overlap, the garbage collector is called.

The garbage collector traces through the stack and all locations in the heap, marking all heap locations containing accessible information, and then compresses the heap by moving all unreclaimable blocks to the low index end. Pointers are adjusted between the marking and compressing phases.

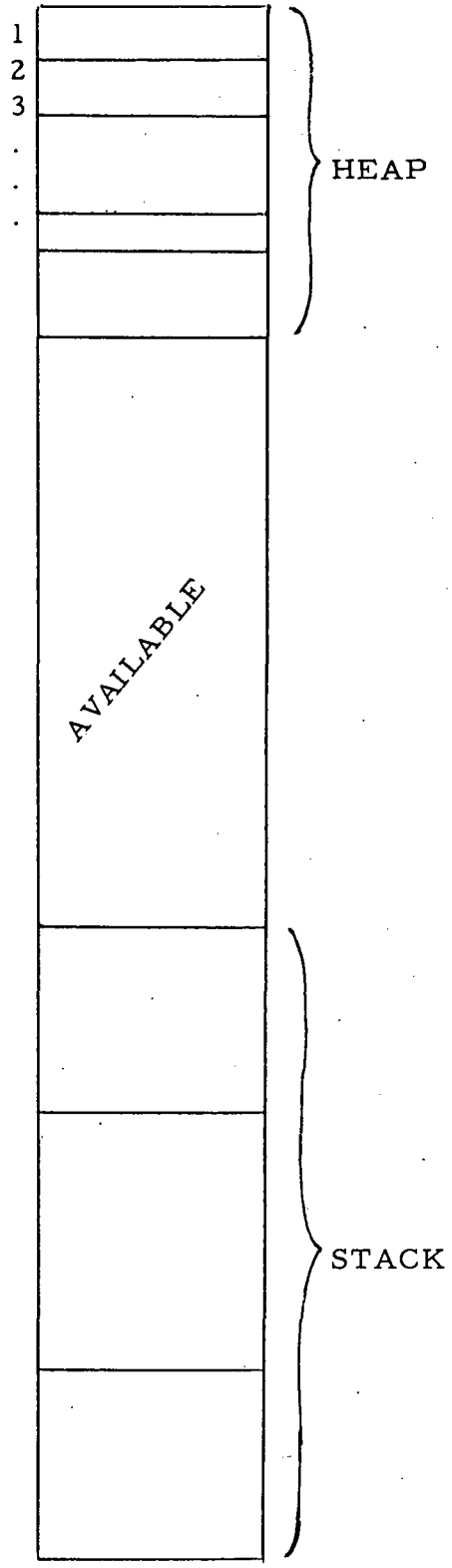
The garbage collector is relatively straightforward, in that a separate table is used to store all "mark" bits, and a fixed size area is used for an auxiliary stack that aids in tracing through data structures.

All data processed by the garbage collector must have the These format are slanted toward SETL application, but are useful for other purposes as well. The garbage collector makes no use of detailed structural information concerning SETL data types. This makes the garbage collector less sensitive to SETL changes and increases its potential for applications having nothing to do with SETL.

All words containing pointers of interest to the garbage collector must follow the basic format shown in the chart below; pointers occur within such words in designated number and in fixed position. ny pointer may be zero, indicating that it doesn't currently point to anything.

DYNAMIC STORAGE

Allocated stack-like, freed by a compacting garbage collector.

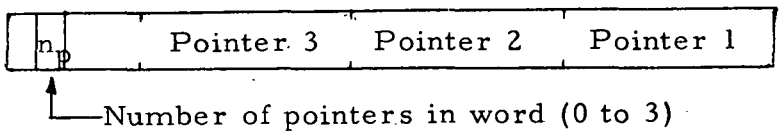


Allocated/freed on subroutine call/return.

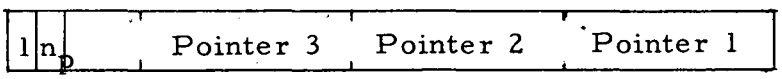
GARBAGE COLLECTOR

WORD FORMATS

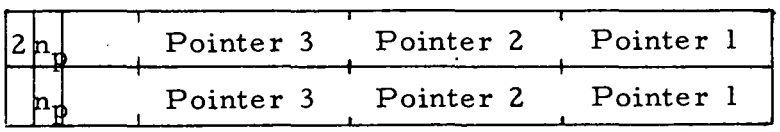
STACK WORD



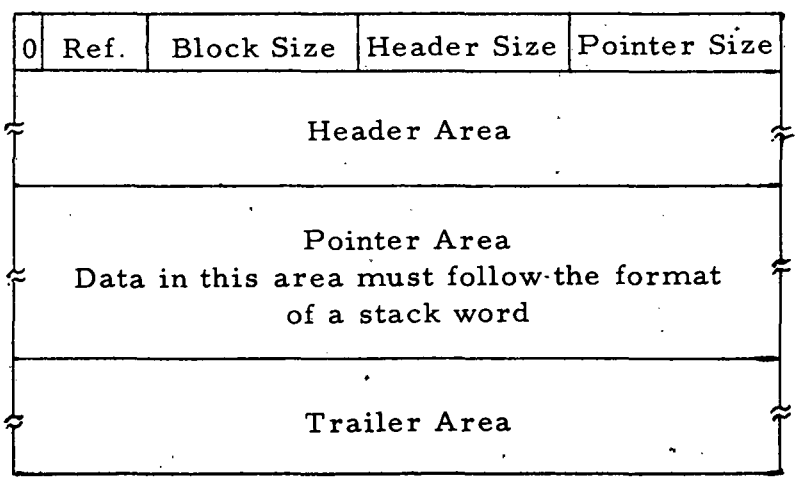
ONE-WORD BLOCK



TWO-WORD BLOCK



STANDARD BLOCK



These types of heap blocks are provided in one-word blocks, two-word blocks, and long blocks of arbitrary size. The type of a block is indicated by its first two bits. The one- and two-word blocks can only contain words of the standard garbage-collector format described above. These small blocks are used for the dense encoding of 'pairs' and other elementary list-nodes.

A 'long block' begins with a header word, which contains information of use to the garbage collector. The remainder of the block is divided into a *header area*, a *pointer area*, and a *trailer area*, each of arbitrary size (including zero). All pointers in the block must be formatted in the standard manner described above and included in the pointer area; all words in this area have the standard garbage collector format. The header and trailer areas are not examined by the garbage collector, and may therefore be used for the storage of floating point numbers, packed character strings, etc.

SETL Data Encoding

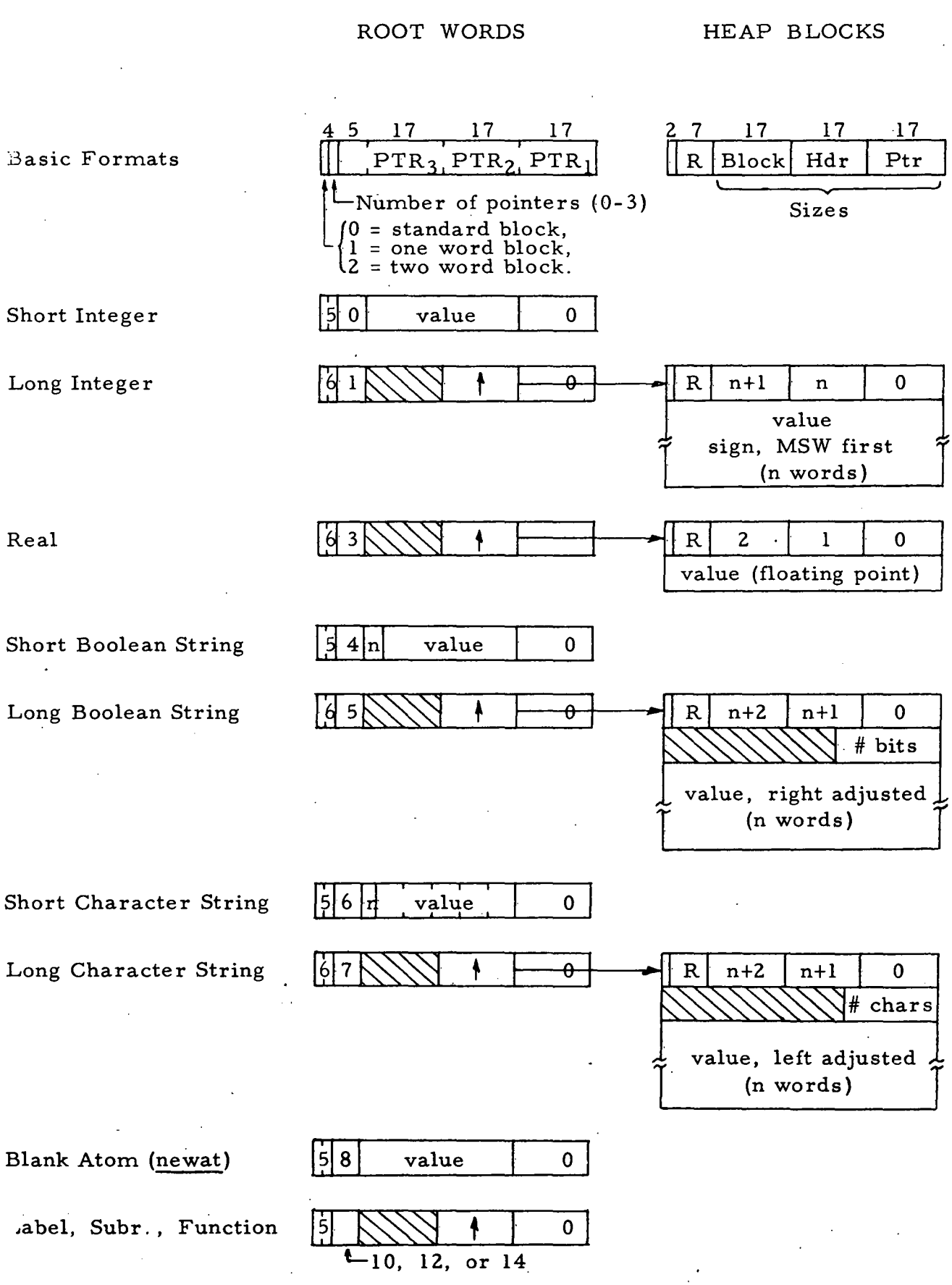
The encoding of SETL data objects is depicted on the chart. Each object is represented by a "root word" that contains two basic fields: *type* and *value*. The value is contained in the root word if it will fit; otherwise the value is stored in the heap, and the root word contains a pointer to the value array. Many items therefore exist in "short" and "long" varieties.

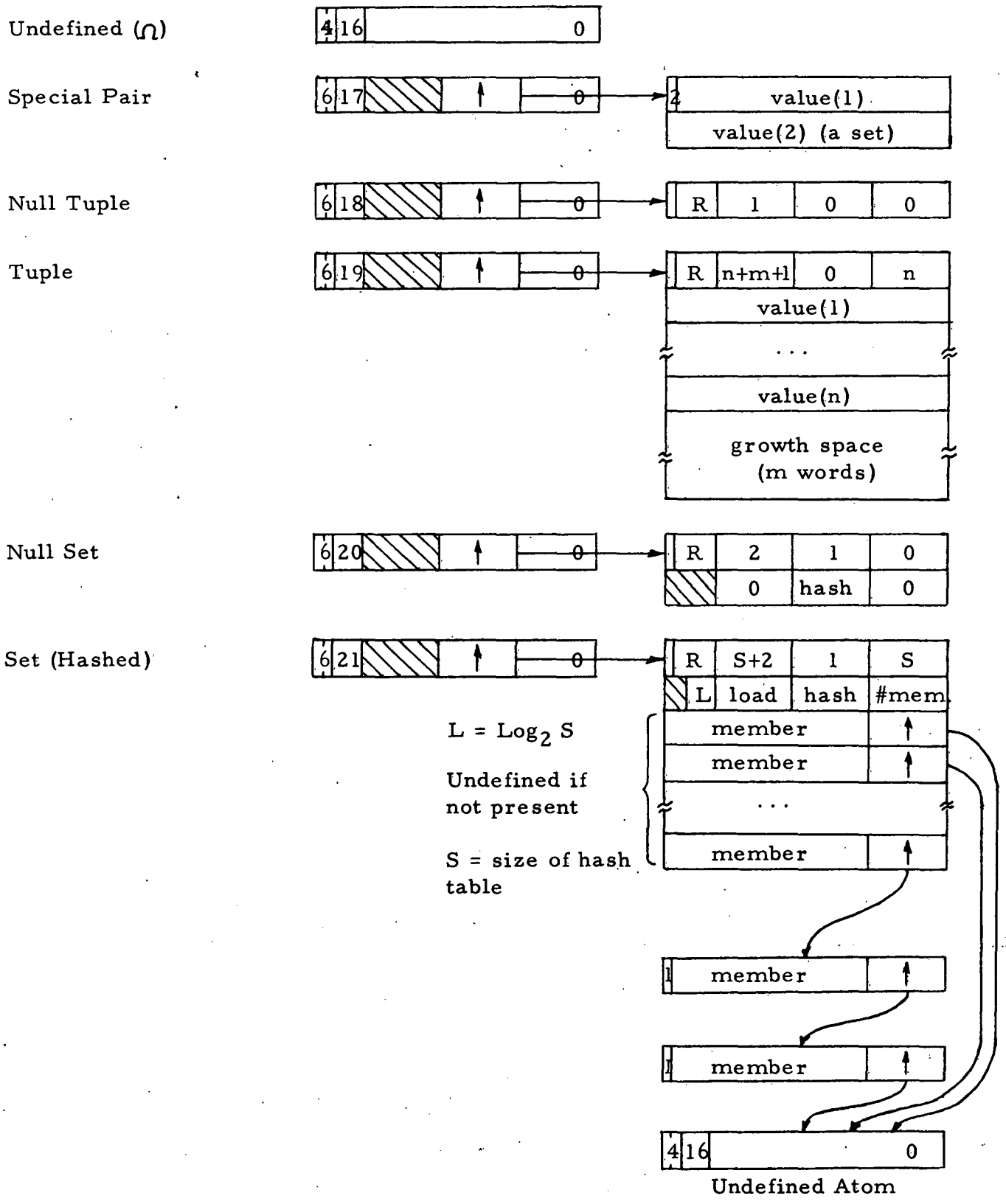
Note SETL allows integers to be arbitrarily large; thus there exist both short and long integers.

"Special pairs" (shown on page 53) do not constitute a SETL data type. They are used in connection with the inclusion of tuples in sets, in a manner described below.

Tuples are provided with a "growth area" node equal to about 25 percent of the tuple's length when space for a tuple is allocated. This is provided so as to allow tuples to grow and shrink at the right-hand ends, an event of frequent occurrence in SETL algorithms.

The null set and null tuple, which play special roles in many SETL run-time library routines, are given unique type codes.


ROOT WORDS
HEAP BLOCKS
Basic Formats
4 5 17 17 17
PTR3 PTR2 PTR1
Number of pointers (0-3)
0 = standard block,
1 = one word block,
2 = two word block.
2 7 17 17 17
R Block Hdr Ptr
Sizes
Short Integer
5 0 value 0
Long Integer
6 1 0
R n+1 n 0
value
sign, MSW first
(n words)
Real
6 3 0
R 2 1 0
value (floating point)
Short Boolean String
5 4 n value 0
Long Boolean String
6 5 0
R n+2 n+1 0
bits
value, right adjusted
(n words)
Short Character String
5 6 n value 0
Long Character String
6 7 0
R n+2 n+1 0
chars
value, left adjusted
(n words)
Blank Atom (newat)
5 8 value 0
Label, Subr., Function
5 0
10, 12, or 14


Undefined (Ω)
4 16 0
Special Pair
6 17 0
2 value(1)
value(2) (a set)
Null Tuple
6 18 0
R 1 0 0
Tuple
6 19 0
R n+m+1 0 n
value(1)
...
value(n)
growth space
(m words)
Null Set
6 20 0
R 2 1 0
0 hash 0
Set (Hashed)
6 21 0
R S+2 1 S
L load hash #mem
L = Log2 S
Undefined if not present
S = size of hash table
member
member
...
member
1 member
1 member
4 16 0
Undefined Atom

The implementation of sets is based on a hashing scheme. Each set is represented by a hash-table structure that contains the set's members. To put an object into a set, a hash code for the object is first calculated. The hash code is used to index the hash table. Set members occupying the same slot of the set's hash table are chained together (each object has a pointer field for this purpose). As long as the lists are short, this scheme allows a reasonably fast implementation of the membership test, even for large sets. Besides membership testing, the operations with (adding a member to a set), and less (deleting a member from a set) are fundamental set operations. Various other operations may be expressed in terms of these. Set equality testing, for example, reduces to a series of membership tests.

The hash table used to represent a set "breathes" as the number of members in the set increases and decreases. Each time a new member (i.e. a member distinct from those already in a set) is added to a set, the current number of members is incremented. The resulting number of members is then compared to the size of the hash table. If the ratio

(number of members) / (hash table size)

exceeds a certain constant, then the hash table's size is doubled. Similar steps are performed when a member is deleted from a set.

The density limits controlling hash-table halving and doubling are set so as to prevent small fluctuations in a set's membership from causing re-sizing of the hash table. Doubling of a hash table is an expensive operation. Space must be reallocated, and the members of the "old" set re-hashed to obtain one more bit of information defining the slot the member should occupy in the new hash table. Halving is also expensive, though somewhat less so.

Salient details of the hashing scheme employed in the present SETL implementation are as follows. Long integers, bit strings, and character strings are reduced to word-length quantities by forming an exclusive or of all their words. The word-length quantities which result are then hashed in much the same way as are inherently short SETL quantities: by combining with an

object-type dependent quantity, and then by dividing by an appropriate constant, to obtain a remainder which is the hash.

The hash code of a tuple is taken to be the hash code of its first component, for reasons that will become clear in the next section. The hash code of a set is the exclusive or of the hash codes of all its members.

With the exception of sets, hash codes are computed on demand, and no attempt is made to save them. The hash code of a set is calculated as the set is built (even if the set is never made a member of another set). This is done so that algorithms dealing with sets of sets can be executed without dismal consequences (our implementation strives to avoid cramping the SETL programmer's style). Each time an element is added to or deleted from a set, the element's hash code (which has of course been calculated) is combined with the set's hash code by an exclusive or operation.

Maintaining precalculated hash codes for sets has the side benefit of allowing these codes to be used in a preliminary comparison during set-equality tests. Sets which are not equal will generally have different hash codes, so that set inequality is recognized rapidly.

Tuples in Sets

Though expressible in terms of the membership test, with, and less operations, functional evaluation plays so important a role in SETL algorithms that we treat it as a primitive.

SETL makes three types of set-related functional evaluation operators available:

```
f(x)
f{x}
f[s]
```

The most fundamental of these is f{x}, which invokes a search over f for all n-tuples that begin with x ($n \geq 2$), and which yields as result the set of all tails of these n-tuples. More precisely, in SETL:

```
f{x} =  if #y eq 2 then y(2) else tl y, y ∈ f |
                type y eq tupl and #y ge 2 and hd y eq x}
```

The operation f(x) has a similar definition but includes a ngle valuedness check:

$$f(x) = \text{if } \#f\{x\} \text{ eq } 1 \text{ then } \ni f\{x\} \text{ else } \Omega \quad .$$

The operation f[s] is adequately defined in terms of f{x}:

$$f[s] = [+: x \in S]\ f\{x\}$$

The fundamental problem in implementing these operations is to provide some method for rapidly locating, within a set, all n-tuples ($n \geq 2$) that begin with a specified component.

Note, as a slight complication, the fact that functional applications with several parameters are allowed. The fundamental definition

$$f\{x,y\} = (f\{x\})\{y\}$$

specifies the semantics of two-parameter functional applications, and similarly for f{x,y,z}, etc. Given a set f containing triples, we may treat f as a function of either one or two variables. That is, both of the expressions f{x} and f{x,y} can be evaluated. A similar remark applies to sets containing quadruples, etc.

Our present implementation supports a reasonably efficient realization of all these functional application operations, at least if the mappings f with which we deal do not contain any very long tuples.

Salient details of this implementation appear in the example shown on the following page, which depicts a set containing six triples and illustrates how it is stored. For the set shown,

$$f\{A\} = \{\langle P,U\rangle,\langle Q,V\rangle,\langle R,W\rangle\}, \text{ and}$$

$$f\{A,P\} = \{U\} \quad .$$

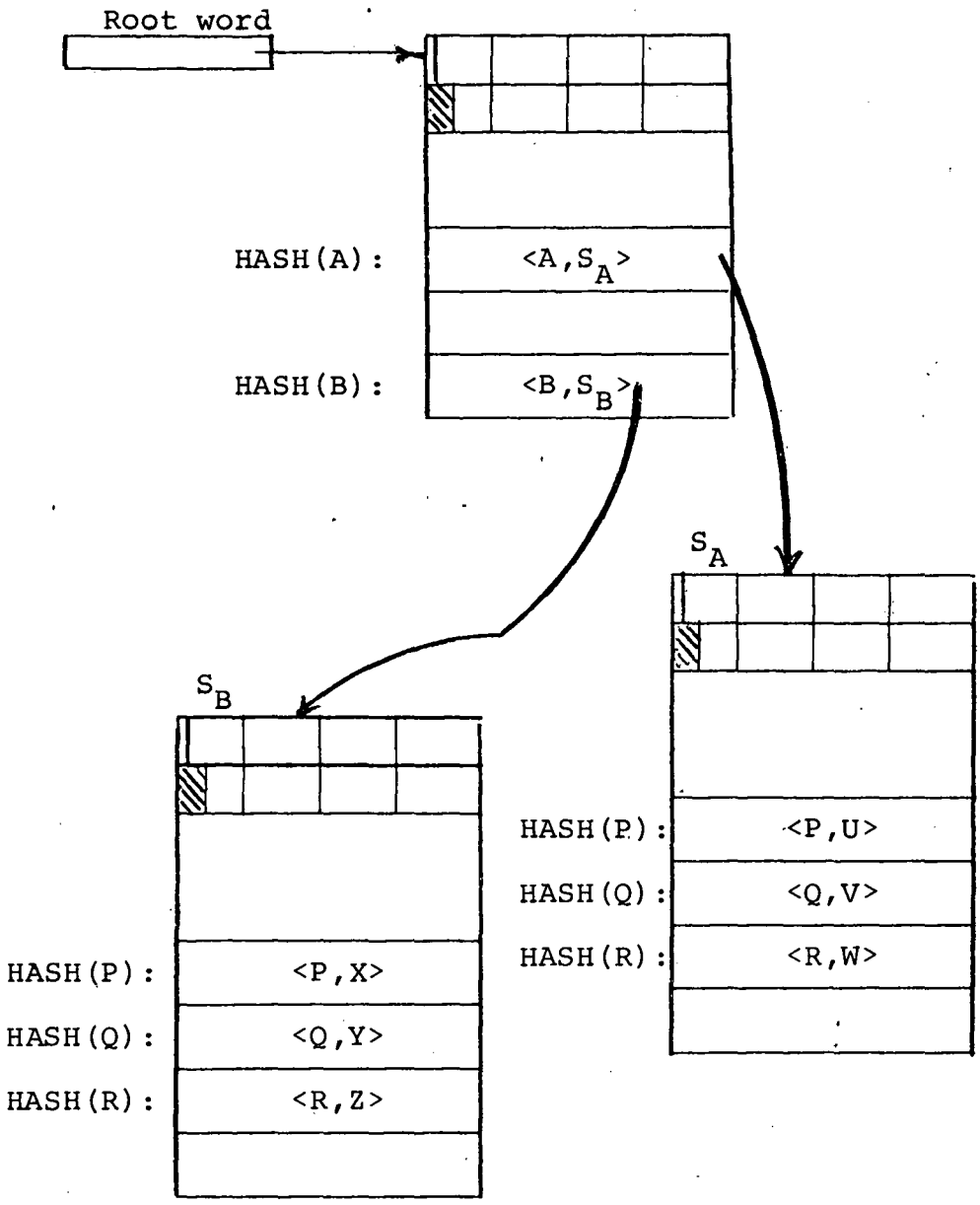
The set's root word points to a *primary hash table* in which are stored two objects that are encoded as "special pairs" (data type 17 of a previous figure). One special pair has as first component the object A, and as second component the set of all tails of n-tuples that begin with A. This is denoted by S_A in the figure. S_A has the same layout as any other set.

TUPLES IN SETS

REPRESENTATION OF

{<A,P,U>, <A,Q,V>, <A,R,W>,

<B,P,X>, <B,Q,Y>, <B,R,Z>}



To evaluate f{A}, the hash code of A is calculated and used as an index into the primary hash table. An overflow list starting at that point is then searched for either a 2-tuple or a special pair beginning with A. If a special pair beginning with A is found, it represents an entire set of items belonging to f{A}, in an already appropriate form. To this set we add the second components of any 2-tuples starting with A which are found.

To evaluate f{A,P}, we first hash A, and then locate a special pair $<A,S_A>$ as before. Then P is hashed, and the secondary hash table S_A is searched for 2-tuples or special pairs beginning with P.

This search process is farily fast but its logic is complicated. The complications will not be dwelt on here, except to observe that if f is

$$f = \{<1,2,3,4>,<1,2,<3,5>>,<1,<2,3,6>>,<1,<2,<3,7>>>\} \quad ,$$

then

$$f\{1,2,3\} \qquad \text{is} \qquad \{4,5,6,7\} \ .$$

Notice that the structure shown in the last preceding chart has objects A and B (or pointers to them) stored only once whereas each of these actually appears three times in an element of the set. Thus our representation affords some storage economy, in lucky cases at least. On the other hand, P, Q, and R are each stored twice.

The above described method of storing tuples in sets is quite poor if one is dealing with a set of tuples that is not used as a function. To take an extreme example, a set containing a single 100-tuple is represented using 99 small hash tables.

Reference Counts

The SETL implementation does not use reference counts to aid or eliminate the garbage collector, but rather in an attempt to avoid unnecessary copy operations. Before discussing implementation details, however, we shall make some remarks stressing the "value oriented" character of SETL.

In the SETL assignment a = b, the object b is (conceptually) copied, and a (conceptually) fresh copy of it becomes the current value of a. This 'logical copying' takes place whether b is 'simple'

or 'compound' in implementation terms, i.e., whether b is an atom, tuple, or a set.

Similarly, when an object is put into a tuple or a set, as in a(3) = b, or s = s with b, it is (conceptually) a copy of b that becomes a member of s or a component of a. Subsequent changes to b do not alter structures or variables into which a former value of b was previously incorporated or assigned.

A crude but logically correct implementation of SETL would always generate copies of data aggregates when they enter into an assignment or are put into other aggregates (PL/1 does this; for example the array assignment A = B expands into a DO loop).

SETL encourages, or at least does not discourage, what might be called a "high level coding sytle" in which one frequently assigns large aggregates to variables, creates maps of aggregates to aggregates, etc., without regard for efficiency questions. To simply generate copies in all these cases would be disastrous both for execution time and for storage requirements.

The issue appearing here may be called the "copy problem". We plan to minimize copying using a strategy with both compile time and run time implications. This optimization is probably the most important optimization for SETL.

Consider the assignment

```
s = s with x;
```

where s is a set and x is a "long" item. In the most efficient implementation of this we "destroy" the copy of s available immediately before the assignment, by putting a reference to x's value (i.e., the root word of x) directly into the hash table representing s. It is often safe to destroy s in this case; i.e. to avoid creating a copy of s either in performing the with operation or in making the assignment.

Consider also the assignment

```
k = i + 1;
```

where i is a long integer. In the most efficient implementation of this operation, we destroy the copy of i available immediately before the assignment, by adding 1 directly to it, and then making k simply point to the result.

To achieve this type of optimization, we will use run time logic associating a reference count field with most objects in the heap. In addition, compile-time live-dead analysis of variables will be used.

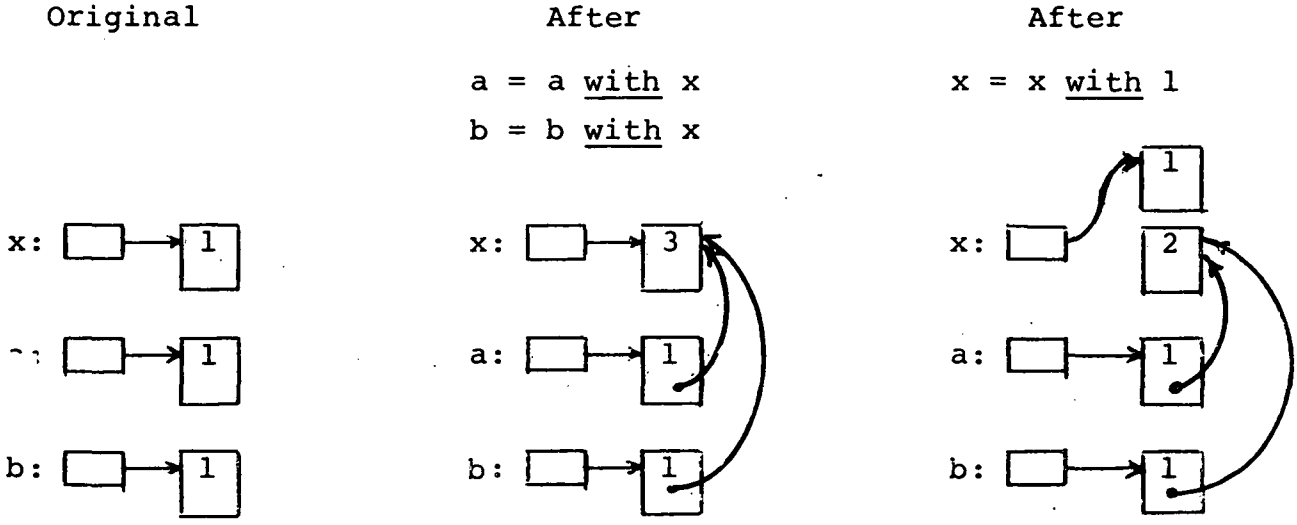
A "live" occurrence of a variable is an occurrence at which there exists a path from the occurrence to a use (i.e., "right-side" context) of the variable. An occurrence of a variable that is not live is "dead"; i.e., all paths from such an occurrence lead either to assignemnts or to program termination.
The reference count associated with an object, which is labeled "R" in a preceding figure, indicates the number of variables or aggregates that point to the object. A value of zero signifies that the space occupied by the object may be collected, but the present garbage collector does not make use of that fact.
(There are two heap items, the one word block and the two word block, that do not have reference count fields. Hence the garbage collector must employ a "bit table" to determine what space is available).

Copy minimization is based on the following rules.

1. Always move only root words on assignment and when putting objects into data aggregates.
2. If a variable is dead and its reference count is one, then the existing copy of it may be updated, even in a manner which will modify it irrevocably.

These rules defer copying until it "must" be done. A concrete example is depicted below.

Figure. Example showing use of reference counts.



In this example we show three sets x, *a*, and *b*; and proceed to make x a member of both *a* and *b*. To evaluate "*a* with x", the compiler generates a call to a set "augmentation" routine, which irrevocably modifies the old value of *a*. This is permissible since this is dead (the value of *a* is immediately changed by the assignment *a* = *a* with x), and since *a*'s reference counter is one. The set x is not copied; instead its reference counter is incremented. Hence in executing the statement *a* = *a* with x no copying of items other than root words is done.

The statement *b* = *b* with x is executed similarly.

Next the program modifies x by adding the integer "1" to it. The compiler will generate a call to the "augment" routine, as before. However, the augment routine will test the set's reference counter before adding the element "1" to it. Since this reference count is greater than one (it is three), the augment routine will first copy x and then proceed to put "1" into the new copy.

This illustrates the compile time and run time activities which eliminate much unnecessary copying in our proposed implementation.

Based on our experience to date with an approximate subset implementation (SETLB) of SETL, we may make a somewhat surprising observation. The SETLB implementation, which is based upon an extended Lisp (BALM) system, does not follow SETL in regard to all details of the semantics of copying. For example, assignments always move only root words and the with routine usually destroys the set being augmented (an exception is made of the null set -- adding a member to it does not destroy the one and only copy of the null set!). Thus the SETLB coder must in principle take care to insert calls to the copy routine in his source program whenever necessary.

Surprisingly, this burden is not very large. Our experience to date shows that in the majority of algorithms it doesn't matter whether a copy is made or not.

Item 8. TECHNICAL PERSPECTIVES.

Our initial experience shows that, as hoped, the SETL system allows us to program complex processes with surprising ease. SETL programs are much shorter than conventional programs realizing the same function; still more, the number of bugs per line of code is sharply decreased from what common experience at the FORTRAN or PL/1 level suggests, and rather straightforward diagnostics connect most bugs closely to their sources. This means that a good part of the greatest obstacle to headway in programming, namely debug time, is in large part overcome by our methods. It now seems clear that a faster running and smaller SETL system, well engineered from the human factors point of view, would allow complex algorithms to be programmed and debugged with remarkable rapidity. It has also become clear that interactive use (which when harnessed to conventional languages does not always yield the degree of advantage which its enthusiasts claim) will undoubtedly be of great benefit when used in connection with a language in which bug symptoms are often quite close to their causes, and in which three-fourths of all bugs can be spotted in a few minutes of inspection.

As part of our own work a substantial number of algorithms, drawn from such areas as optimization, parsing, and theorem-proving, have been written and debugged in the language we call SETLA. The SETLA Users Manual (Item 11) describes this language and lists a few debugged algorithms. It is of course difficult to quantify the relative difficulty of programming the same problem in different languages, but SETLA users, comparing SETLA and FORTRAN, report ratios such as 1/10.

Our experience to date as users of the presently available prelimintary version of SETL (a bulky and very slow system) convinces us that the labor of programming can be reduced at least to 1/5 of current levels or better by providing a polished SETL environment fast enough to eliminate turnaround delays.

Holding this basic view, our overall aim is to demonstrate what we think can in time become new widely accepted programming technique: the writing of programs in a very powerful language having the abstract flavor of SETL, followed by its optimization by a combination of automatic and programmer-assisted procedures. We aim to demonstrate this technique, first as a technology for program design, and subsequently as a technology of program implementation.

We have realized from the start of our work that programming by the use of very high level, mathematicized dictions will at first imply a substantial loss of efficiency. (See below for estimates.) We have hoped that by gaining a deeper understanding of the optimization problems which arise in connection with languages of high level, and by applying our new dictional tools to the specification and design of the required optimizers, a good part of this loss can be made up. Our two years of work have in fact clarified the problems which must be faced in this connection. These problems now seem quite soluble, though by no means easy. A significant step in this direction is the development of the Data Structure Elaboration Language discussed below. However, success in our immediate objective of attempting to demonstrate a system must of course rest on the levels of efficiency which we are able to achieve in the relatively short run. Here our work is far enough along that we are able to predict the following levels of performance with a good degree of confidence. (Note that here and below we refer to FORTRAN simply to be able to name a typical low-level language of good efficiency.)

Predicted efficiency of SETL system presently under construction:

Spe d, as fraction of FORTRAN speed	3 %
Space required, as multiple of FORTRAN array size	6/1

These efficiency levels obviously imply severe limitations on the programming load supportable on presently available hardware. Nevertheless, they will allow our new programming techniques to be demonstrated on a scale substantial enough to be convincing. To get a feel for what is implied, we consider two types of job: fir a 'module debug' job, in which a fragmentary group of subroutines

(equivalent in logical function to a few hundred FORTRAN statements) ; being tested against a very small sample data set; secondly a ;ystem integrate' job in which many routines (constituting, for example, an optimizing compiler) are being tested together. The first type of job we define as requiring 0.4 sec. (6600 time, exclusive of compile and I/O time) if executed in FORTRAN; the second type of job we assume will require 2 min. if executed in FORTRAN. Runs of the first type will use 12 seconds (of 6600 time) in our new SETL system. Runs of the second type will require 1 hour.

We define an ideal debugging environment as an interactive system capable of providing a programmer with 1 run every 5 minutes through a 4-hour debug session. This is 48 runs/day; on the 6600, these runs would consume about 10 minutes/day of processor time. Six programmers steadily at work in a 'module debug' mode (probably equivalent to a group of 18 with access to the system) will therefore consume 1 hour/day. 'System integrate' debugging will of course impose much more substantial burdens on available computational resources. Approximately 18 runs/shift are probably ideal for this type of work; the 6600 is capable of providing only 1/2 this much. On a dedicated 6600, two projects operating in 'system integrate' mode (one day shift, one night shift) would probably coexist acceptably.

Faster machines would proportionally increase the sustainable load; the following figures are suggestive.

	Supportable 'Module Debug' Population	*Supportable 'System Integrate' Projects*
CDC 6600 (3 megacycles)	18	2
CDC 7600 (12 megacycles)	72	6
Hypothetical 100-megacycle computer	600	50

The last alternative describes an active national algorithms laboratory; the second a much more modest level of activity, but still one that should amply demonstrate the specification technology we are in process of developing. A 7600 system would probably provide a level of service which could be rented to organizations

tempting to develop large and difficult systems; even a dedicated --00 system is too small to permit more than a minimum demonstration of our techniques at the 'system integrate' level.

It is worth noting that the vigorous development of new higher-speed hardware is the most straightforward, if also the most expensive, way of getting into a position in which the availability of new programming techniques can be clearly demonstrated. There is at the present time nothing particularly difficult about building a 100 megacycle machine; however, the capable manufacturers lack the will to do so.

The technical picture sketched above only becomes complete if we add to it a discusson of the amounts of memory required to support the envisaged debugging and integration activity.

The bulk of compiled SETL will consist of calling sequences. Examination of typical code of this kind reveals the following rules of thumb. Each 'active token' in SETL source must compile into at least 4 bytes of code at the machine level. We count as active tokens each variable name and operator symbol appearing in the source; it is reasonable to count '(' as an active symbol, but to regard the matching ')' as a possible mark required for syntactic purposes only. Four bytes are about minimal, since 2 bytes or more will normally be required for an operand or code sequence address, 1 byte additional for opcode information, and at least 1 byte additional for miscellaneous overhead connected with machine-level housekeeping. Tokens average out to about 3 characters, and 15-20 tokens make the average SETL card. Thus we must compile about 80 bytes/card, and in fact will probably wind up compiling 160 bytes/card. By the same rough reckoning, FORTRAN compiles as 3 bytes/token, and about 16 bytes/card. Since 10 FORTRAN statements have roughly the logical function of one SETL line, code sizes in SETL programming should run at roughly par with code sizes of programs of equivalent functions written in FORTRAN, except that a substantial initial block of code, namely the 'run time library' realizing the SETL operations, forms part of the SETL memory requirements. These considerations lead to the following rough formula for the anticipated size of SETL runs, in which we write FCODE for FORTRAN code size, FDAT for FORTRAN data sizes (in bytes).

$$SETLSIZ = 240{,}000 + FCODE + 6 * FDAT \quad .$$

This establishes 300,000 bytes, or approximately 120K (octal 6600 machine words), as the minimum size of a run in the next SETL system. Software paging should reduce core requirements somewhat, possibly making minimum runs in less than 100K possible.

It is also worth considering the space which would be required for a large 'integration' run. Assume, for example, a program 4,000 SETL cards long, probably equivalent in logical function to a 40,000 card FORTRAN program (e.g., the whole of an optimizing compiler). The FORTRAN program is taken to require 320,000 (decimal) bytes of data space. Applying the above formula we estimate a SETL size of 2,800,000 bytes, or 350,000 (decimal) 6600 machine words. Jobs of this type could therefore actually be run on CDC 7600 computers furnished with large amounts of bulk core.

The figures just given indicate that one more step of hardware development is all that is necessary for the methods we propose to become quite practical, at least as techniques for the specification and rapid prototyping of complex systems. For an entirely convincing demonstration to be made using present equipment, we require some technique which increases efficiency considerably without reimposing an over-heavy burden of programming. The preliminary theoretical studies which we have carried out indicate that it ought to be possible to meet demand by developing a number of optimization techniques, including in particular a Data Structure Elaboration Language (DSEL, explained below). We estimate that the use of this optimization technique should approximately double programming effort (as compared to the use of 'pure SETL') but should improve efficiency to the extent shown in the following figures.

Predicted efficiency of SETL with use of DSEL

Speed, as a fraction of FORTRAN speed	20%
Space required, as multiple of FORTRAN array size	2/1

These estimates point up a particularly exciting technical possibility. One-fifth of the speed of FORTRAN is a level of performance quite acceptable even now for a wide range of commercial applications.

It will be made still more acceptable by the hardware developments currently under way. We therefore consider that a perfected SETL/DSEL combination will constitute a 'new software technology', usable broadly in the commercial marketplace, which could more than cut in half the labor generally required for applications programming.

However, full development of the optimization methods which can culminate in the DSEL requires the solution of conceptual and design problems which we have not yet fully unraveled, and will also require a substantial programming effort. The availability of a SETL *cum* DSEL system therefore lies some years in the future. As a stopgap we therefore need some more readily accessible technique for bringing critical programs to a usable level of efficiency. Something of what we desire can very probably be achieved by allowing a 'mixed style' in which the bulk of a program is written in SETL, but certain portions vital for efficiency (in the sense either of run-time or of data size) are rewritten directly in the LITTLE implementation language. Since programs generally spend more than 90% of their time executing less than 10% of their code, this could bring us to 20% of FORTRAN speed and within a factor of 3 of FORTRAN data sizes, still with a 40% reduction in programming effort as compared to conventional technique. In comparison, we expect the perfection of the DSEL, a much more difficult task, to attain the same speed, to reduce data space requirements to not more than twice FORTRAN sizes, and to allow a 60% reduction in programming labor. Moreover, a DSEL system would in many cases eliminate most of the 240,000 byte SETL run-time library code from the final compiled form of SETL programs, though the SETL/DSEL compiler that accomplished this would undoubtedly be substantially larger than a simple SETL compiler.

Data Structure Elaboration Language (DSEL)

An important issue which we will eventually have to face is connected with our notion of a *data structure elaboration language:* There is an inevitable conflict between optimization and interactiveness. High-quality optimization requires that extensive global analysis should be applied to a program, and that the information

gained by this analysis should be used to transform the program extensively and globally. Such a process leads one via a complex sequence of decisions to a tightly integrated optimized code. However, for debugging, easy modifiability and quick incremental patching are most desirable; to this end, the more loosely bound a system, the better. This conflict of approaches may at a later stage in our work force us to develop two closely related but quite distinct SETL systems, one for debugging, the other incorporating all the optimization techniques which we are able to discover.

A similar tension exists between two possible approaches to program optimization. Optimization is ideally fully automatic; automatic optimization has the obvious advantage of imposing no optimization-related labor on the user of a programming language. We have already begun to use our improved programming tools to explore fully automatic optimization techniques in as penetrating a way as we can. However, we are presently of the view that the SETL level of abstraction may leave open too wide a choice of combinations of data structures for fully automatic optimization to reach successfully from this level of description to the choice of truly optimal data structures. On the one hand, this indicates the success of part of our original SETL plan, namely, to allow a programmer to postpone the choice of data structures without foreclosing any important structural possibilities. Our success however leaves open so total a range of possibilities as to imply that the fully automatic optimization of SETL programs into efficient code is probably a very difficult problem. On the basis of these reflections we have decided not to employ only fully automatic but also programmer-assisted optimization techniques in connection with SETL.

The data structure elaboration language will stand at the center of our approach to programmer assisted optimization. This language is essentially a language of declarations; inclusion of appropriate DSEL statements in a running SETL algorithm will not affect its results but will improve its efficiency substantially. The information specified by DSEL statements is essentially that inherent in the initial 'data structure design' which is an important initial

consideration for programmers using conventional techniques. However the DSEL allows this design to be described formally; conventional this essential information plays only an informal role. In a conventional approach, the programmer holds in his mind both a data structure design and a set of algorithms, and compiles these manually into a detailed code. In the approach toward which we are working, a formal data structure design and a debugged algorithm will be submitted to a compiler which will then produce all necessary code.

It is important in contemplating what a data structure elaboration language can accomplish to have a clear understanding of the general nature of the transformations which a programmer conventionally applies to an algorithmic plan in order to arrive at a detailed data structure design. We see the following as basic to this process: those objects which are to be used as 'indices', i.e., which are elements of the domain of one or more mappings, are issued 'serial numbers' (the serial number of an object is generally the position within some hash table or other array at which the 'actual object' is stored). The values of mappings defined for such objects are then retrieved by direct indexing using the serial numbers of objects, rather than by the considerably less efficient repeated hashing which SETL ordinarily requires. Moreover a programmer will normally enhance the efficiency of a program by exploiting such information concerning the abstract objects appearing in it as the fact that all values of a function may lie in a very small range, etc. The DSEL we project will allow both patterns of indexing and information of this latter sort to be declared, and, provided we are able to master the optimization-related problems which it raises, should make possible the very significant increases in the efficiency of SETL which have been indicated above.

Concerning our approach to the choice of semantic facilities and detailed syntax for the SETL language, the following deserves to be noted. By deliberate choice we are restricting the present version of SETL to a subset of the full collection of facilities which our design studies have shown to be desirable. This subset language, essentially our present SETLB, is sufficiently powerful to furnish a clear demonstration of the new programming methods

toward which we are working. In addition, brought to an acceptable design point of speed and size, it furnishes us with the tool we need to develop a second, fuller, version of SETL, and also to attack the optimization problems outlined in the preceding paragraphs.

As a byproduct of our experimentation with programming technique in a wide range of application areas, we expect to produce a fairly comprehensive library of algorithms, written in SETL and actually debugged. We expect this library to be broadly useful. The algorithms which it is to contain will represent many of the processes of central interest in graduate computer science instruction, such as compilers, optimizers, processes for grammar analysis and transformation, table compression, sorting algorithms, algorithms in the artificial intelligence area, and the like. It is our intent to use this library not only as a basic text for study by students wishing to learn how important processes are programmed, but also as a mechanism allowing them to experiment with a much larger family of processes than would otherwise be accessible to them. Moreover, these algorithms will be of use as prototypes to groups undertaking major software developments, ourselves included.

We may also note that even that light use of our present system which we have till now been able to make convinces us that our 'multiphase' programming technique, with its orderly progression from executable algorithm specification language to efficient lower level code will lead to the writing of *better* programs than ordinarily are produced by present technique. To write good final code, one must really begin with a penetrating contemplation of the algorithmic approaches suitable to a given problem, choose one to be implemented, bring it to logical completion as a debugged abstract program, and then, proceeding carefully at each stage, make the following steps:

i. Review the text of the abstract program initially written, searching for simplifications of method, more natural and more modular descriptions of the same function, and improvements in function desirable for generality or from a human-factors viewpoint.

Revise and re-debug the abstract program, to bring it to a high standard in all these regards.

ii. Consider, in as many variants as necessary, data structures into which the abstract program resulting from (i) can be mapped in a manner implying concrete efficiency; make whatever software measurements are necessary for the relative advantages of each particular data structure to become clear. Choose that particular design which appears from such study to be optimal; record it, using a formal or semiformal language like the SETL DSEL to do so.

iii. Using an appropriate implementation language, code the concrete algorithm specified by (i) and (ii), in this process paying particular attention to the realization of those 'innermost' processes critical for efficiency.

The realities of present day programming make it rare indeed for all the steps of the deliberate approach outlined above to be accomplished accurately. Our new techniques should make it much easier to reach this degree of accuracy, and should in this sense lead to an overall improvement in the quality of programming. The SETL algorithm library which we have begun to accumulate will be an important first stage in this process, since it will come to consist of carefully thought out, debugged abstract programs on which concrete implementations can be built.

Item 9. A PRECIS OF THE SETL LANGUAGE.

In the present section, we summarize the principal basic features of the SETL language, as they have been defined in the preceding pages. It is hoped that this *precis* can serve as a useful brief reference.

Basic Objects: *Sets* and *atoms*; sets may have atoms or sets as members. Atoms may be

Integers	Examples: 0, 2, -3
Real	Examples: 9., 0.9, 0.9E-5
Boolean strings	Examples: 1b, 0b, 77b, 00b777
Character strings	Examples: 'aeiou', 'spaces- '
Label (of statement)	Examples: label:, <label:>

Blank (created by function newat). Ω is special 'undefined' atom.
Subroutine. Function.
The operator type x returns the type of the object x.

Basic operations for atoms:

Integers: arithmetic: +, -, *, /, // (remainder)
comparison: eq, ne, lt, gt, ge, le
other: max, min, abs
Examples: 5//2 is 1; 3 max -1 is 3; abs -2 is 2.
Reals: Above arithmetic operations (with exception of //) plus exponential, log, and trigonometric functions.
Booleans: logical: and (or a), or, exor, implies (or imp), not (or n)
logical constants: t (or true, or 1b); f (or false, or 0b).
Character strings: conversion: dec, oct
Examples: dec '12' is 12; oct '12' is 10.
Strings (character or boolean):
+ (catenation), * (repetition), a(i:j), a(i:) (extraction), # (size), nulb, nulc (empty strings).
Examples: 'a' + 'b' is 'ab'; 2 * 1b4 is 11001100b; 2 * 'ab' is 'abab', 'abc'(1:2) is 'ab', 'abc'(2:2) is 'bc', 'abc'(2) is 'b', # 'abc' is 3, # nulc is 0.

General: Any two atoms may be compared using eq or ne;
atom a tests if a is an atom.

Basic operations for sets:

ϵ (membership test); nl (empty set); ∋ (arbitrary element);
(number of elements); eq, ne (equality tests);
incs (inclusion test); with, less (addition and deletion of element);
lesf (ordered pair deletion); pow(a) (set of all subsets of a);
npow(k,a) (set of all subsets of a having exactly k elements).
+ (set union), * (intersection), // (symmetric difference).

Examples: a ϵ {a,b} is t, a ϵ nl is f, ∋ nl is Ω,
∋ {a,b} is either a or b, #{a,b} is 2, # nl is 0,
{b} with a is {a,b}, {a,b} less a is {b},
{a,b} less c is {a,b}, {a,b} incs {a} is t.
pow({a,b}) is {nl,{a},{b},{a,b}}.
npow(2,{a,b,c}) is {{a,b},{a,c},{b,c}}.

Tuples

Ordered tuples are treated as SETL objects of different type than sets -- e.g. tuples may have some components undefined.

Operations on tuples:

Tuple former: If x,y,...,z are n SETL objects then
t = <x,y,...,z> is the n-tuple with the indicated components.
#t is the number of components of t
t(k) is the k-th component of t
t(i:j) is the tuple whose components, for $1 \le k \le j$, are t(i+k-1)
hd t is t(1)
tl t is t(2:)
+ is the concatenation operator for tuples

Examples: hd <a,b> is a. tl <a,b> is <b> which is not the same object as b. If t = <a,b> and τ = <a,c> then

T = t + τ = <a,b,a,c> T(3:2) = <a,c>

Tuple components may be modified by writing

t(j) = x;

An additional component may be concatenated to t by writing

t(#t + 1) = x;

Set-Definition: by enumeration: {a,b,...,c} . Set-former:

$\{e(x_1,\ldots,x_n),\ x_1 \in e_1,\ x_2 \in e_2(x_1),\ldots,x_n \in e_n(x_1,\ldots,x_{n-1})$
$\mid C(x_1,\ldots,x_n)\}$.

The *range restrictions* x ∈ a(y) can have the alternate *numerical form*

$$\min(y) \leq x \leq \max(y)$$

when a(y) is an interval of integers.

If t is a tuple, the form x(n) ∈ t can be used, see below, *iteration headers*, for additional detail.

Optional forms include

{x ∈ a | C(x)} equivalent to {x, x ∈ a | C(x)}; and

{e(x), x ∈ a} equivalent to {e(x), x ∈ a | t} .

Functional application (of a set of ordered pairs, or a programmed, value-returning function):

f{a} is {if #p gt 2 then tl p else p(2), p ∈ f | if type p ne tupl then f else (#p) ge 2 and (hd p) eq a}, i.e.
is the set of all x such that <a,x> ∈ f .

f(a) is: if # f{a} eq 1 then ∍f{a} else Ω,
i.e., is the unique element of f{a}, or is undefined atom.

f[a] is the union over x ∈ a of the sets f{x}, i.e., the *image* of a under f.

More generally:

f(a,b) is g(b) and f{a,b} is g{b}, where g is f{a};

f[a,b] is the union over x ∈ a and y ∈ b of f{x,y} .

If f is a value-returning function, then

f{a,b} = {f(a,b)}, f[a] = {f(x), x ∈ a}, etc.

Constructions like f{a,[b],c}, etc. are also provided.

Compound operator:

[op: x ∈ s]e(x) is $e(x_1)$ op $e(x_2)$ op ... op $e(x_n)$,

where s is $\{x_1,\ldots,x_n\}$.

This construction is also provided in the general form

[op: $x_1 \in e_1$, $x_2 \in e_2(x_1)$,...,$x_n \in e_n(x_1,\ldots,x_{n-1})$ | $C(x_1,\ldots,x_n)$]e(x) ,

where the range restrictions may also have the alternate numerical form, or the form appropriate for tuples.

Examples: [max; x ∈ {1,3,2}](x+1) is 4,

[+: x ∈ {1,3,2}](x+1) is 9,

[+: x(n) ∈ a]x is SETL form of $\sum_{i=1}^{n} a$

[op: x ∈ nl]e(x) is Ω .

Quantified boolean expressions:

∃x ∈ a | C(x) ∀x ∈ a | C(x)

general form is

$\exists x_1 \in a_1,\ x_2 \in a_2(x_1),\ \forall x_3 \in a_3(x_1,x_2),\ldots \mid C(x_1,\ldots,x_n)$

where the range restrictions may also have the alternate numerical form, or the form appropriate for tuples.

Evaluation of

∃ x ∈ a | C(x)

sets x to first value found such that C(x) eq t.

If no such value, x becomes Ω.

The alternate forms:

min ≤ ∃x ≤ max, max ≥ ∃x ≥ min , max ≥ ∃x > min, x(n) ∈ t, etc.

of range restrictions may be used to control order of search.

Conditional expressions:

if $bool_1$ then $expn_1$ else if $bool_2$ then $expn_2$ ... else $expn_n$.

a orm b abbreviates if a ne Ω then a else b

a andd b abbreviates if n a then f else b

Statements: are punctuated with semicolons.

Assignment and multiple assignment statements:

a = expn; f{exp} = expn; is the same as

f = {p ∈ f | (hd p) ne exp} + {<exp,x>, x ∈ expn};

f(exp) = expn; is the same as f{exp} = {expn};

f(a,b) = expn; f{a,b} = expn; etc. also are provided.

<a,b> = expn; is the same as a = expn(1); b = expn(2);

<a,b,...,c> = expn; <a,<b,c>,...,d> = expn; etc. are also provided.

<f(a),g{b}> = expn; is the same as f(a) = expn(1); g{b} = expn(2);

Generalized forms:

<f(a), g{b,c},..., h(d)> = expn;

<f(a),<g{b,c},h(d)>,...,k(e)> = expn; etc. also are provided.

Use of general expressions on left-hand side of assignment statements (sinister calls).

$e(x_1,\ldots,x_n) = y;$ must be no-op if executed immediately after $y = e(x_1,\ldots,x_n);$ and vice-versa. The use

op op' x = y;

of a product operator on the left-hand side of an assignment expands as

```
t = op'x;
op t = y;
op'x = t;
```

with similar rules for multiparameter compounding. These rules allow user-defined functions to be used quite generally on the left-hand side of assignment statements. The 'left hand' significance of a function is often deducible from its standard right-hand side form, but may be varied by using specially designated code blocks which are executed only if the function is called from right-hand or left-hand position respectively. These have the respective forms:

(load) block;	(execution only if function called from right-hand side of assignment)
(store x) block;	(execution only if function f called is from $f(param_1,\ldots,param_n) = x;$).

Commonly used operators having special side effects:

expn is x	has same value as expn and assigns this value to x
x in s;	same as s = s with x;
x from s;	same as x = ɔs; s = s less x;
x out s;	same as s = s less x;

Use of code blocks within expressions.

If *block* is a section of text which could be the body of a function definition, then [; block] is a valid expression which both defines and calls this function. Such code block expressions can be used freely within other expressions.

Control statements

go to label;

if $cond_1$ then $block_1$ else if $cond_2$ then $block_2$...else $block_n$;

if $cond_1$ then $block_1$ else ... else if $cond_n$ then $block_n$;

Iteration headers

(while cond) block;

(while cond doing blocka) block;

is equivalent to (while cond) block blocka;

$(\forall x_i \in a_1, x_2 \in a_2(x_1), \ldots, x_n \in a_n(x_1, \ldots, x_{n-1}) \mid C(x_1, \ldots, x_n))$ block;

in this last form, the range restriction may have such alternate numerical forms as

$\min \leq x \leq \max$, $\max \geq x \geq \min$, $\min \leq x < \max$, etc.,

which control the iteration order.

If t is a tuple, bit string, or character string, then the operator of form (∀x(n) ∈ t) block; is available. This is an abbreviation for

$(1 \leq \forall n \leq \#t \mid t(n) \text{ ne } \Omega)x = t(n);$ block;

Iterators of this form may also be used in set formers, compound operators, quantifiers, etc.

Iterator Scopes

The scope of an iteration or of an *else* or *then* block may be indicated either with a semicolon, with parentheses, or in one of the following forms:

end ∀; end while; end else; end if; etc.;

or: end ∀x; end while x; end if x; etc.

Loop control

quit; quit ∀x; quit while; quit while x;

and

continue; continue ∀x; continue while; continue while x;

The *quit* statement terminates an iteration; the *continue* statement begins the next cycle of an iteration.

Subroutines and functions (are always recursive)

To call subroutine:

sub(param$_1$,...,param$_n$);

sub[a]; is equivalent to (∀x ∈ a) sub (x);;

Generalized forms:

sub(param$_1$,[param$_2$,param$_3$],...,param$_k$)

are also provided.

To define subroutines and functions:

subroutine:

```
define sub(a,b,c); text  end sub;
return;  -- used for subroutine return
```

function:

```
definef fun(a,b,c); text  end fun;
return val;  -- used for function return
```

infix and prefix forms:

```
define a infsub b; text  end infsub;
definef a infin b; text  end infin;
define  prefsub a; text  end prefsub;
definef prefun  a; text  end prefun;
```

Namescopes

Scope declarations divide a SETL text into a nested collection of scopes. Scope names are known in immediately adjacent, containing, and contained scopes. Other than this, names are local to the scope in which they occur, unless propagated by include or global statements.

Declaration forms

scope name; ... ; end name;

scopes with specified numerical level

scope n name; ..., end name;

global declaration

global name$_1$, ..., name$_n$;

with specified numerical level

global n name$_1$,..., name$_n$;

nclude statement

include list$_1$, ..., list$_n$;

Example:

include bigscope1(scope1 x,scope2(-z),scope3(x,y,u[v])),bigscope2*

'*' signifies all elements known in scope, '-' signifies exclusion of those elements listed, [ ] modifies the 'alias' under which an element is known in scope in which included. Subroutines and functions are scopes of level 0. Macros (see below) are transmitted between scopes in much the same way as variable names. The declaration

owns routname$_1$(x_1,...,x_{n1}), routname$_2$(y_1,...,y_{n2}), ...

states that the variables x_j are stacked when *routname*$_1$ is entered recursively, the variables y_j are stacked when *routname*$_2$ is entered recursively, etc.

Macro blocks

To define a block: macro mac(a,b); text endm mac;

To use: mac(c,d);

Initialization

initially block; (*block* executed only first time process entered)

Input-Output

Unformatted character string:

A SETL file is a pair <st,n>, where st is a character string and n an integer indexing one of its characters.

er is end record character; input, output are standard I/O media; the function record(s); -- reads a file <st,n> from position n till er character or string-end is encountered in the character string *st*.

Standard format I/O

An interval file f in SETL is a pair <st,n> consisting of a character string *st* and an index n to one of the characters of *st*.

f read name$_1$,...,name$_n$; using standard format reads from file <st,n>, starting at position n

f print expn$_1$,...,expn$_n$; using standard form transfers external representation of objects to file s = <st,n>, starting at position n as above.

The set {s_1,...,s_n} is represented as {r_1,...,r_n}, where r_j is the external representation of s_j. Similarly, the tuple <s_1,...,s_n> is represented as <r_1,...,r_n>.

An external file *st* in SETL is character string catalogued with the operating system supporting SETL under some identifying name *catname* (which is itself a string). The statement

```
x = open catname;
```

makes the string *st* into the value of x. The call

```
close(st,catname)
```

makes the SETL string *st* into the contents of the external file named by the string *catname*.

Item 10. CORRESPONDENCES BETWEEN SETL AND SETLA.

SETLA, the presently implemented SETL subset, differs in a number of regards from the 'official' SETL described in Item 9 of the present work. We now give a few remarks, which we hope will help clarify the correspondence between the SETL and SETLA languages. The text of the SETLA Users Manual then follows as the final item of the present manuscript.

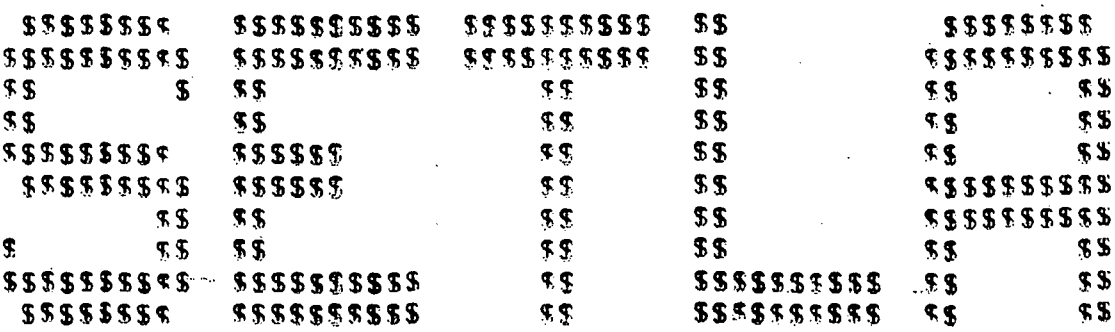
Item (in SETL Precis)	Comment
character strings	SETLA users the ≠ sign for quotation marks
Boolean operators	Neither exclusive or nor implies are included as SETLA operators
atom	In SETLA, the TYPE function is used instead of this predicate.
set functions	In SETLA, arbitrary element (ARB.) is the first element of a set, in an implementation defined order. it is not random.
quantified boolean expressions	See p. 97 and also p. 126 of SETLA Users Manual.
code-blocks within	Supported in SETLA, but with slightly variant syntax. Cf. the formal grammar, pp. 121-124 in the SETLA Users Manual.
assignment statements	See p. 97 and p. 108 of SETLA Users Manual for extent of implementation.
control statements	Review the formal grammar pp. 121-124 of SETLA Users Manual to see extent of implementation. Conditional statements and condition expressions are included.

loop control	*quit* and *continue* are not implemented in SETLA.
name scopes	In SETLA, name scopes are external (global) unless declared local. Moreover, they are handled dynamically rather than statically.
include and local declarations	Not implemented in SETLA. See BEGIN and LOCAL statements, p. 98 of SETLA Users Manual.
macros	See p. 109 of SETLA Users Manual. To use, in SETLA, write: mac(C,D); To define in SETLA write +*mac(A,B) = text**
initialization	Not implemented in SETLA.
compile function	Not explicitly implemented in SETLA, but available through BALM 'incremental compile' feature.
input-output	See p. 126, 127 for input-output in SETLA.

Item 11. SETLA USERS MANUAL

SETL NEWSLETTER NR, 70

AUG,01,1974
J,SCHWARTZ
S, BROWN
E, SCHONBERG



SETLA USERS MANUAL
VERSION 1,0-JAN,1975

TABLE OF CONTENTS

0. INTRODUCTION

SETLA IS AN IMPLEMENTED VERSION OF A SUBSET OF SETL, A SET-THEORETIC LANGUAGE DEVELOPED AT NYU BY JACK SCHWARTZ. A DETAILED DESCRIPTION OF SETL CAN BE FOUND IN "ON PROGRAMMING", VOLS. I AND II. SETLA IS FOR THE MOST PART A COMPATIBLE SUBSET OF SETL (TO A MUCH GREATER EXTENT THAN ITS PREDECESSOR, SETLB). HOWEVER, BALM STILL PLAYS AN IMPORTANT ROLE IN THE STRUCTURE OF SETLA, AND THIS IS REFLECTED IN SOME OF ITS FEATURES WHICH DO NOT PROPERLY BELONG TO STANDARD SETL. NOTE IN PARTICULAR THAT :

NAME SCOPING AND THE SPECIAL CONVENTIONS WHICH APPLY TO BEGIN BLOCKS AND STATEMENT LABELS ARE THE SAME IN SETLA,

BALMSETL, AND, INDEED, IN BALM. THE SETLA PRECEDENCE RULES AND PROCEDURE LINKAGE MECHANISMS ARE HOWEVER THOSE OF SETL, RATHER THAN THOSE OF BALM OR BALMSETL.

"BALMSETL" ABOVE REFERS TO A SERIES OF BALM PROCEDURES WHICH IMPLEMENT SETL PRIMITIVES AS BALM EXTENSIONS. THE STRUCTURE, AND INDEED THE EXISTENCE OF BALMSETL OUGHT TO BE INVISIBLE TO THE SETLA USER, AND WILL BE FOR THE MOST PART, SEE HOWEVER SEC. 3, 5 AND 8 FOR UNAVOIDABLE INSTANCES OF BALMSETL VISIBILITY.

THIS MANUAL IS NOT INTENDED TO PROVIDE A DESCRIPTION OF SETL PER SE. THE READER IS ASSUMED TO BE FAMILIAR WITH THE FIRST TWO ITEMS IN "ON PROGRAMMING", VOL II.

SETLA SUPPORTS A FEW EXTRA STATEMENTS, NOT BELONGING TO STANDARD SETL. THESE PROVIDE VARIOUS BALM-LIKE FEATURES - IN PARTICULAR, INCREMENTAL EXECUTION AND BALM BEGIN-END BLOCKS. THESE STATEMENTS WILL BE EXPLAINED IN DETAIL BELOW.

1. LEXICAL CONVENTIONS, MACROPROCESSOR

THE LEXICAL CONVENTIONS OF SETLA ARE ESSENTIALLY THOSE SET FORTH IN "ON PROGRAMMING- VOLII, WITH SOME SLIGHT CHANGES THAT CAN BE GLEANED FROM THE FOLLOWING SECTION. NOTE IN PARTICULAR THAT :

A. "UNDERLINED" NAMES OF STANDARD SETL ARE REPRESENTED BY THE CORRESPONDING PERIOD-TERMINATED NAMES IN SETLA.
B. SET BRACKETS ARE ≤ AND ≥ ,
C. THE MEMBERSHIP OPERATOR IS REPRESENTED BY ↠ .

THE SETLA PREPROCESSOR INCORPORATES THE LITTLE MACROPROCESSOR, AND THEREFORE SUPPORTS MACROS LIKE THOSE OF LITTLE, CF. THE GUIDE TO THE LITTLE LANGUAGE FOR DETAILS.

MACROS WITH PARAMETERS ARE DECLARED IN THE FORM :

```
** MACRONAME(ARG1,ARG2,..,ARGK) = MACROBODY **
```

MACROS WITH NO ARGUMENTS ARE DECLARED IN THE FORM :

```
** MACRONAME = MACROBODY **
```

FOR EXAMPLE :

```
**  HDHDTL = HD. HD. TL. **
**  SUMM(X. Y. Z) = (X + Y + Z) **
```

THESE MACROS CAN BE INVOKED AT ANY POINT IN THE PROGRAM BY WRITING :

```
MACRONAME(SUB1, SUB2,..,SUBK)
```

OR SIMPLY:

```
MACRONAME
```

IN THE CASE OF A MACRO WITHOUT ARGUMENTS. FOR EXAMPLE:

```
V(I) = HDHDTL SUMM(A, TL.B, C) ;
```

WILL EXPAND INTO THE FOLLOWING STATEMENT :

```
V(I) = HD.HD.TL.(A + TL.B + C) ;
```

2. SETLA FUNCTIONS, CONSTANTS AND OPERATORS

MEANING	SETLA	REMARKS
TYPE FUNCTION	TYPE,	
ATOM PREDICATE	ATOM,	
TYPES:		
INTEGER	INT,	
REAL	REAL,	
BLANK	BLANK,	
SET	SET,	
TUPLE	TUPL,	
CHARSTRING	STR,	
LABEL	LAB,	
BOOLEAN	BITS,	
SUBROUTINE	SUBR,	
CONSTANTS:		
NULLSET	NL,	
NULL-STRING	NULC,	
NULL-TUPLE	NULT,	
TRUE	T,	
FALSE	F,	
UNDEFINED	OM,	
COMPARISON AND BOOLEAN OPERATORS		
EQUALS	EQ,	
NOT EQUAL	NE,	
LESS THAN	LT,	
LESS-EQUAL	LE,	
GREATER	GT,	
GREATER-EQUAL	GE,	
INCLUDES	INCS,	SET-THEORETIC INCLUSION, APPLIES TO BOOLEANS AND BIT-STRINGS,
IMPLIES	IMP,	
AND	A,,AND,	
OR	O,,OR,	
NOT	N,,NOT,	

FOR BIT-STRINGS OF ARBITRARY LENGTH, THE FOLLOWING FUNCTIONS

ARE PROVIDED :

AND	LAND(X,Y)
OR	LOR(X,Y)
EXCLUSIVE OR	XOR(X,Y)
COMPLEMENT	LNOT(X)

ARITHMETIC OPERATORS

PLUS	+
MINUS	-
TIMES	*
DIVIDE	/
RESIDUE	//
EXPONENTIATION	EXP.
MAXIMUM	MAX.
MINIMUM	MIN.
ABSOLUTE VALUE	ABS.

SETLA SUPPORTS ARBITRARY PRECISION INTEGER ARITHMETIC, I.E. THE ARITHMETIC OPERATORS CAN BE APPLIED TO INTEGERS OF ARBITRARY SIZE.

CHARACTER STRING OPERATIONS

DECIMAL CNVT	DEC.	
CATENATE	+	CATENATE AND REPEAT ARE
REPEAT	*	ALSO PROVIDED FOR BIT-STRINGS.
SUBSTRING	C(I,J)	EXTRACT J ITEMS, STARTING WITH I-TH
		SUBSTRING AND LENGTH ALSO
LENGTH	#	APPLY TO BIT-STRINGS AND TUPLES.

THE EXTERNAL REPRESENTATION OF ANY ATOM X IS RETURNED BY THE FUNCTION STRINGOF(X).

TUPLE OPERATIONS :

TUPLE-FORMER	<X1,...,XN>
HEAD	HD.
TAIL	TL.
COMPONENT	T(I)
LENGTH	#
CATENATION	+

```
SET OPERATIONS
--- ----------
  MEMBERSHIP          X→A
  NUMBER              #
  WITH                WITH,
  LESS                LESS,
  LESF                LESF,
  DIMINISH            X OUT,S
  ALGMENT
  DIMINISH-F          X OUTF,S
  UNION               +
  INTERSECTION        *
  DIFFERENCE          -
  SYM.DIFF.           //
  ARB, ELEMENT        ARB,S


  SET OF GIVEN
      ELEMENTS        {!X1,...,XN}

NOTE DISCREPANCY BETWEEN SETLA AND SETL.
IN SETL, SETS BY ENUMERATION TAKE THE FORM :
                      {X1,...,XN}


  POWERSET            POW(S)
  ALL N-ELEMENT       NPOW(N,S)
     SUBSETS


GENERAL SET-FORMER:
  { E(X),X→A+C(X)}
  { E(X),X→A}
  { E(N), 1<N<K}
  { E(X,N),X→A, 1<=N<=M+ C(X,N) }

THE SIMPLE SET-FORMER EXPRESSION :

  { X ,X→A+C(X) }

CAN BE ABBREVIATED AS :

  {  X→A+C(X) }
```

FUNCTIONAL APPLICATION

APPLICATION	F(X)
MULTIVALUED APPLICATION	F≤X≥
RANGE	F[X]

THE CORRESPONDING FORMS EXIST FOR FUNCTIONS OF SEVERAL VARIABLES

F(X1,...,XN)
F≤X1,...,XN≥
F[X1,...,XN]

ITERATION STATEMENTS :
---------- ----------

```
(∀X∈S) BLOCK ;
(∀X∈S↑C(X))BLOCK;
(∀M<=K<=N)BLOCK;
(∀M<=K<=N ↑C(K))BLOCK;
```

ALL MEANINGFUL COMBINATIONS OF COMPARISON OPERATORS CAN BE USED TO SPECIFY AN ARITHMETIC RANGE, FOR EXAMPLE,

```
(∀M<K<=N)BLOCK;
```
AS WELL AS
```
(∀M>=K>=N)BLOCK;
```

NOTE DISCREPANCY BETWEEN SETLA AND SETL:
IN SETLA, THE ITERATION SYMBOL APPEARS IN FRONT OF THE RANGE SPECIFIER, AND NOT IMMEDIATELY PRECEEDING THE ITERATION VARIABLE,

SETLA AND SETL ALSO PROVIDE COMPOUND ITERATORS
```
(∀X∈S,M(X)<=Y<=N(X)↑C(X,Y))BLOCK;  ETC,
```

WHILE-ITERATORS,

```
(WHILE G)BLOCK;
(WHILE G DOING BLOCK1)BLOCK2;
```

-CONTINUE- AND -QUIT- STATEMENTS CAN BE USED , BOTH WITHIN -FORALL- AND -WHILE- ITERATION LOOPS, THEIR ACTION HOWEVER, IS LIMITED TO THE LOOP THAT CONTAINS THEM; JUMPS OUT OF SEVERAL LOOPS AT ONCE ARE NOT ALLOWED, EITHER KEYWORD MAY BE FOLLOWED BY UP TO 4 TOKENS CORRESPONDING TO THE LOOP OPENER, IN ACCORDANCE TO THE USUAL SETL RULES, SEE SEC,3 ON ITERATOR SCOPES FOR DETAILS;

QUANTIFIED BOOLEAN EXPRESSIONS:

```
  ∃X→S|C(X)       EXISTENTIAL QUANTIFIER
  ∃M<= K <=N|C(X)
```

ON EXIT FROM AN EXISTENTIAL SEARCH-LOOP, THE QUANTIFIED VARIABLE IS ASSIGNED TO THE FIRST OBJECT IN THE SPECIFIED RANGE WHICH SATISFIES CONDITION C. IF NO SUCH OBJECT IS FOUND, THE QUANTIFIED VARIABLE HAS VALUE OM.

```
  ∀X→S|C(X)       UNIVERSAL QUANTIFIER
  ∀M<=K<=N|C(X)
```

COMPOUND FORMS FOR BOTH QUANTIFIERS ARE SUPPORTED :

```
  ∃X→S, Y→F(X) | C(X,Y)
  ∀X→S, Y→G(X), U(Y)<=Z<=W(X,Y) | C(X,Y,Z)
```

FOR COMPOUND EXPRESSIONS THAT USE BOTH QUANTIFIERS, THE SECOND QUANTIFIER SHOULD APPEAR AS PART OF THE EXPRESSION FOLLOWING THE " SUCH THAT" SYMBOL, FOR EXAMPLE :

```
  ∀X→S, ∃Y→F(X) | C(X,Y)          MUST BE WRITTEN AS :
  ∀X→S | (∃Y→F(X) | C(X,Y))      TO WHICH IT IS CLEARLY EQUIVALENT.
```

CONDITIONAL EXPRESSIONS

```
  X=IF Y→S THEN 1 ELSE 2;
```

NOTE: TO AVOID AMBIGUITIES, THIS CONSTRUCT SHOULD BE PATENTHESIZED WHENEVER IT IS PART OF A MORE COMPLICATED EXPRESSION. E.G. :

```
X= (IF Y→S THEN A ELSE B) + C ;
```

NOTE THAT THE "ELSE" CLAUSE IS REQUIRED.

COMPOUND OPERATORS:
--------- ---------

```
[+:X→S] F(X)
[*:X→S  * C(X)] F(X)
[+:1<N<=M,  Y→S(N)*C(N,X)]F(X,N)
[MIN,:X→Y]F(X)                    ,ETC.
```

SINISTER FORMS:
--------- -----

```
NAME          = EXPRESSION
HD, NAME      = EXPRESSION;
TL, NAME      = EXPRESSION;

NAME(PARAM1,...,PARAMK) = EXPRESSION;
NAME≤PARAM1,...,PARAMK≥ = EXPRESSION;
NAME[PARAM1,...,PARAMK] = EXPRESSION;
```

THESE SINISTER FORMS ARE EVALUATED FROM RIGHT TO LEFT; I.E. THE "EXPRESSION" IS EVALUATED FIRST, THEN THE INDICES, AND FINALLY THE OBJECT BEING ASSIGNED TO, THIS DEPARTURE FROM THE STANDARD LEFT-TO-RIGHT EVALUATION ALLOWS THE CONSTRUCTION OF EXPRESSIONS WHERE EVALUATION OF RHE R.H.S. OF THE ASSIGNMENT (THANKS TO A RECURSIVE CALL ,SAY) MAY ACTUALLY MODIFY THE OBJECT BEING ASSIGNED INTO.

```
NAME(PARAM1,...,PARAMK) = OM,;
NAME(N1:N2) =EXPRESSION;
```

NOTE : THIS LAST CONSTRUCT IS NOT A GENERAL SUBSTRING REPLACEMENT FUNCTION, IT REQUIRES THE LENGTH OF THE REPLACEMENT SUBSTRING TO BE EQUAL TO N2.

GENERALIZED SINISTER ASSIGNMENTS, OF THE FORM :

```
<F(A),G(1), HD,Y> = TUP ;
HD, TL, HD, X = Y ;
```

ARE NOT CURRENTLY SUPPORTED.

FUNCTION,SUBROUTINE, AND OPERATOR DEFINITION;

```
DEFINE SUB(PARAM1,...,PARAMK);
DEFINEF FNC(PARAM1,...,PARAMK);
DEFINE  MON, PARAM;                  MONADIC FORMS
DEFINEF MON, PARAM;
DEFINE  P1  BIN, P2;                 BINARY FORMS
DEFINEF  P1 BIN, P2;
DEFINE  NOARGOP,;                    FORMS WITH NO ARGUMENTS
DEFINEF NOARGOP,;
RETURN;                             -RETURN FROM SUBROUTINE
RETURN(EXPRESSION)                  -RETURN VALUE FROM FUNCTION
```

INPUT =OUTPUT;

```
PRINT, EXPN1,...,EXPNK;
READ, NAME1,...,NAMEK;
WRITE,FILENO, NAME1,...,NAMEK;
```

SEE SEC.8A FOR DETAILS OF CURRENT I-O IMPLEMENTATION.

MISCELLANEOUS SYSTEM FUNCTIONS.

NEWAT, — RETURNS A NEW IDENTIFIER, OR BLANK ATOM, AT EACH INVOCATION.

RANDOM(X) — IF X IS AN INTEGER, THIS FUNCTION RETURNS A PSEUDO-RANDOM INTEGER IN THE RANGE (1,X) . IF IT IS A STRING ,A TUPLE OR A SET, A RANDOM ELEMENT IS RETURNED.

MISCELLANEOUS ADDITIONAL STATEMENT FORMS:

```
EXPN IS, S          -ON THE FLY- ASSIGNMENT ; HAS THE
                     SAME VALUE AS EXPN, AND ASSIGNS THIS
                     VALUE TO S.
A IN, S ;           EQUIVALENT TO  S = S WITH,A ;
A OUT,S ;           EQUIVALENT TO  S = S LESS,A ;
A FROM,S ;          EQUIVALENT TO:
                        A = ARB,S ; A OUT,S ;

FINISH;        -REQUIRED TERMINATOR FOR COMPLETE SETL PROGRAM
NOOP;          -NO OPERATION

RETN;          -RETURN FROM A  CODE BLOCK (SEE SEC.4),
                  EQUIVALENT TO THE BALM/BALMSETL
                                    RETURN();
LOCAL NAMEA,NAMEB,....
               -DECLARES LOCAL VARIABLES WITHIN A
                  SUBROUTINE OR FUNCTION,  IF USED,
                  SHOULD FOLLOW IMMEDIATELY ON
                  SUBROUTINE/FUNCTION DEFINITION HEADER,

DO;            -STARTS A BALM/BALMSETL DO GROUP,
                  THE ELEMENTARY UNIT OF EXECUTION.
                  SHOULD BE TERMINATED BY
COMPUTE ;
```

3.STATEMENT LABELS, ITERATOR SCOPES, ITERATION SCOPE END-MARKS:

A LABEL IS A NAME FOLLOWED BY A COLON, A STATEMENT MAY BE PREFIXED BY ANY NUMBER OF LABELS, E.G.

```
LABEL: X=Y;
LABEL1: LABEL2:  Y=Y+1;
```

IMPORTANT :

NOTE THAT IN SETLA (AS IN SETLB) LABELS ARE RECOGNIZED BY THE BALM SYSTEM ONLY IF THEY ARE -NOT- WITHIN THE SCOPE OF A -FORALL- OR -WHILE- ITERATOR, THIS MEANS THERE MUST BE NO UNCLOSED SCOPES BETWEEN THE DEFINITION OF A LABEL AND THE PROCEDURE DEFINITION STATEMENT, FOR EXAMPLE,

```
DEFINE SUBA(X);
IF X EQ, 1 THEN GO TO LAB1;;
PRINT, ≠CASE1≠;
LAB1: PRINT, ≠THIS LABEL OK≠;
RETURN;
END;
```

IS OK, SINCE THE DEFINITION OF -LAB- IS IN NO SCOPE, HOWEVER,

```
DEFINE SUBA(X);
(~ 1<= NTIMES <=5)
IF X EQ, 1 THEN GO TO LAB1;;
PRINT, ≠CASE1≠;
LAB1: PRINT, ≠THIS LABEL OK≠;
END ~ 1< NTIMES;
RETURN;
END;
```

WILL NOT WORK, SINCE THE LABEL IS DEFINED WITHIN A FORALL LOOP

ITERATOR SCOPES:

A SCOPE IS OPENED BY:
1. A ≠FORALL≠ ITERATOR
2. A ≠WHILE≠ ITERATOR
3. A SUBROUTINE OR FUNCTION DEFINITION ;

EACH SUCH SCOPE MUST BE CLOSED BY A CORRESPONDING END-ELEMENT, WHICH MUST HAVE ONE OF THE FOLLOWING FORMS:

A. AN EXTRA SEMICOLON
B. ≠END ;≠
C. ≠END≠, FOLLOWED BY UP TO 4 TOKENS OTHER THAN SEMICOLON, FOLLOWED BY A SEMICOLON,

EXAMPLES:

```
(~X→S) X=X;;
(~X→S) X=X;END ~X;
(WHILE X→S DOING  X=X+1;) Y=X; END  WHILE X→S;
DEFINE A  OP, B; ....... END A OP.;
```

IF A SCOPE IS ENDED IN THE FORM C, DESCRIBED ABOVE, THE EXISTENCE OF A SCOPE OPENER MATCHING THE SCOPE ENDER WILL BE VERIFIED, AND ANY UNCLOSED SCOPES FOLLOWING THE LAST-OPENED MATCHING OPENER WILL BE CLOSED, WITH APPROPRIATE ERROR MESSAGES BEING GIVEN.

NOTE: THAT THE MANNER IN WHICH A SUBROUTINE OR FUNCTION IS ENDED DIFFERS SLIGHTLY FROM THE CONVENTION APPLICABLE TO OTHER SCOPES, SINCE NEITHER ≠DEFINE≠ NOR ≠DEFINEF≠ SHOULD APPEAR AMONG THE FOUR OPTIONAL TOKENS FOLLOWING ≠END≠, THE FOLLOWING ARE VALID :

```
DEFINEF FN(X)    ; ....... END;
DEFINEF FN(X)    ; ....... END FN( ;
DEFINE SUB(X,Y)  ; ....... END SUB ;
DEFINE SUB(X,Y)  ; ....... END SUB(X,Y );
DEFINE X OP,Y    ; ....... END X OP, ;
```

4. ADDITIONAL MISCELLANEOUS SYNTACTIC INFORMATION:

OPERATOR PRECEDENCE , FUNCTIONAL APPLICATION, CODE BLOCKS WITHIN EXPRESSIONS:

OPERATOR PRECEDENCE RULES IN SETLA ARE THE SAME AS IN SETL. THE USER SHOULD CONSULT ≠ ON PROGRAMMING≠ ,VOL II FOR DETAILS.

ONLY TWO OPERATOR PRECEDENCE LEVELS ARE USED IN SETLA: A,VARIOUS BOOLEAN-VALUED COMPARISON OPERATORS,TO WIT

EQ,,NE,,GT,,GE,,LT,,LE,,→,INCS,,IMP.

WHICH BIND MORE STRONGLY THAN OTHER OPERATORS

B, OTHER OPERATORS ASSOCIATE TO THE LEFT

SETLB USERS PLEASE NOTE : THESE PRECEDENCE RULES ARE THE ONES INTENDED FOR SETL, AND NOT THE ONES PREVIOUSLY IN EFFECT IN SETLB.

MONADIC OPERATORS HAVE MINIMAL SCOPE, THUS
-A+B MEANS (-A)+B, N, X A,Y MEANS (N,X) A,Y
NOTE: +S EQ, 0 MEANS +(S EQ, 0)
SINCE BOOLEAN-VALUED OPERATORS BIND MOST STRONGLY.
SIMILARLY, THE EXPRESSION A+B GT, C+D
WILL BE PARSED AS A+(B GT,C)+D

USERS SHOULD KEEP THESE PRECEDENCE RULES IN MIND, AS THEY DIFFER SOMEWHAT FROM THOSE OF OTHER PROGRAMMING LANGUAGES. WHEN IN DOUBT, PARENTHESIZE.

THE FOLLOWING EXAMPLES WILL ILLUSTRATE THE PRECEDENCE RULES DESCRIBED ABOVE:

A-B+C	MEANS (A-B)+C
-A+B	MEANS (-A)+B
∃X→S+F(X)A,Y	MEANS (∃X→S+F(X)) A,Y
[+:X→S]G(X)+Z	MEANS ([+:X→S]G(X))+Z
X A,Y NE,W	MEANS X A,(Y NE, W)

```
X NE,Y A,W        MEANS  (X NE,Y) A, W

≤:<1,2>,<2,3>,<3,4>≥ (2)
    IS A  LEGITIMATE ≠MAP APPLIED TO PARAMETER≠
    EXPRESSION, AND HAS THE VALUE 3. SIMILARLY,

≤:<1,2>,<1,3>,<2,4>,<3,5>≥ [<1,3>]
    IS LEGITIMATE, AND HAS THE VALUE
       ≤:2,3,5≥
```

WARNING : IN SETLA, THE OPERATOR -IS,- IS TREATED AS ANY OTHER INFIX OPERATOR, AND DOES NOT FOLLOW THE PRE- CEDENCE RULES SPECIFIED FOR IT IN C.P. VOL II,P.20.

CODE BLOCKS WITHIN EXPRESSIONS:

SETLA,LIKE SETL, ALLOWS A BLOCK OF CODE TO BE USED AS PART OF AN EXPRESSION, BOTH FOR THE VALUE IT RETURNS AND FOR THE SIDE-EFFECTS WHICH ITS EVALUATION MAY CAUSE. A BLOCK OF CODE USED IN THIS WAY BEGINS WITH THE SYMBOLS [: AND ENDS WITH THE SYMBOL]. SUCH A BLOCK SHOULD CONTAIN AT LEAST ONE STATEMENT OF THE FORM :

```
RETN EX;
```

WHERE -EX- DENOTES ANY EXPRESSION, ONE OF THESE STATEMENTS SHOULD BE THE LAST STATEMENT EXECUTED WITHIN ITS BLOCK, THE VALUE OF -EX- THEN DEFINES THE VALUE OF THE ENTIRE BLOCK

EXAMPLE:

```
A=A+[:X=0;(WHILE F(X) LT, Z) X=X+F(X);; RETN X;];
```

5. NAME SCOPING, PROCEDURE LINKAGE.

SETLA IS AN EXTENSION OF BALM, HENCE THE NAMESCOPING RULES ARE THOSE OF BALM.
THE SETL *EXTERNAL* STATEMENT IS NOT CURRENTLY SUPPORTED.

THE FOLLOWING BASIC FACTS ABOUT BALM NAMESCOPING RULES SHOULD BE NOTED :

A GLOBAL VARIABLE IS NOT DECLARED EXPLICITLY, IT IS KNOWN AT THE OUTERMOST PROGRAM LEVEL AND INSIDE ALL BLOCKS IN WHICH THE SAME NAME HAS NOT BEEN DECLARED AS A LOCAL VARIABLE.

A LOCAL VARIABLE IS DECLARED WITHIN A PROCEDURE BLOCK BY ENTERING ITS NAME IN A LIST IMMEDIATELY FOLLOWING THE KEYWORD "LOCAL". A LOCAL VARIABLE IS KNOWN IN THE PROCEDURE IN WHICH IT IS DECLARED AND IN ALL PROCEDURES WHICH ARE CALLED FROM THAT PROCEDURE AND WHICH DO NOT CONTAIN A DECLARATION OF A LOCAL VARIABLE WITH THE SAME NAME.

THE STATEMENT

```
LOCAL  NAMEA,NAMEB,... ;
```

MUST APPEAR AS THE FIRST STATEMENT WITHIN THE SUBROUTINE BODY. SUCH A STATEMENT CAUSES THE VARIABLES IN THE NAME-LIST WHICH IT CONTAINS TO BE LOCAL TO THE SUBROUTINE/FUNCTION WITHIN WHICH IT APPEARS; THESE VARIABLES WILL, IN PARTICULAR, BE STACKED/UNSTACKED ON SUBROUTINE ENTRY/RETURN.

THE SUBROUTINE DEFINITION

```
DEFINE F(X);...BODY...;END;
```

IS TRANSLATED INTO THE PARSE-TREE FOR THE BALM EXPRESSION

```
F= PROC(X),BEGIN(),...TRANSLATION OF BODY...()END END.
```

IF A LOCAL STATEMENT APPEARS, AS IN

```
DEFINE F(X);LOCAL A,B ;...;END;
```

THE PARSE-TREE PRODUCED CORRESPONDS TO :

```
F=PROC(X),BEGIN(A,B),...TRANSLATION OF BODY...()END END.
```

CALLING CONVENTIONS OF SETLA ARE THOSE OF SETL.
THEY DIFFER FROM THOSE OF BALM AND SETLB. SETLA LINKAGE MECHANISMS PROVIDE CALL BY VALUE WITH DELAYED ARGUMENT RETURN.
EXECUTION OF A PROCEDURE CALL ENTAILS THE FOLLOWING :

1.- ON ENTRY TO A PROCEDURE, THE CALLING PARAMETERS ARE COPIED INTO A LOCAL BLOCK RESERVED FOR THE PROCEDURE-S FORMAL ARGUMENTS, AND AN IDENTIFIER FOR EACH CALLING PARAMETER IS CREATED AND SAVED, THIS IDENTIFIER REFERENCES THE ENVIRONMENT IN WHICH EACH CALLING PARAMETER RESIDES. IT WILL TAKE ONE THE FOLLOWING FORMS :
   A.- A SYMBOL TABLE ENTRY (FOR GLOBAL VARIABLES.)
   B.- A POINTER INTO THE ENVIRONMENT BLOCK OF THE CALLING PROCEDURE (FOR LOCAL VARIABLES OR FORMAL ARGUMENTS OF THE CALLING PROCEDURE).
   C.- UNDEFINED. (FOR EXPRESSIONS).
2.- UPON RETURN FROM THE PROCEDURE, THE CURRENT VALUES OF THE FORMAL ARGUMENTS ARE COPIED BACK INTO THEIR ENVIRONMENTS, BEFORE RETURNING CONTROL TO THE CALLING PROCEDURE.

THE USER MAY REFER TO SETL NEWSLETTER 60 FOR FURTHER DETAILS.
IT FOLLOWS THAT FUNCTIONS AND SUBROUTINES CAN HAVE SIDE-EFFECTS ON GLOBAL VARIABLES AS WELL AS ON THEIR CALLING PARAMETERS, FOR EXAMPLE

```
DEFINE EFFECT(X,Y,Z,T,U) ;
X = 0 ;
Y = NL1 ;
Z(1) = 5 ;
T = T WITH,U ;
U = 2*U ;
END ;

A = 20 ;
B = 0 ;
C = <1,2,3> ;
D = NL1 ;
E = ≠ABCDE≠ ;

EFFECT(A,B,C,D,E) ;
PRINT(A,B,C,D,E) ;
```

WILL PRODUCE THE FOLLOWING OUTPUT

```
0 NL1 <5,2,3> <≠ABCDE≠>     ≠ABCDEABCDE≠
```

NOTE HOWEVER, THAT ARGUMENT RETURN ONLY TAKES PLACE FOR ATOMIC ARGUMENTS, AND NOT FOR GENERAL EXPRESSIONS THAT MAY BE VALUE-RECEIVING,FOR EXAMPLE, IF EXECUTION OF PROCEDURE SUB(X,Y) MODIFIES ITS ARGUMENTS, THEN THE CALL

```
SUB(Z,S(V))
```

WILL MODIFY Z, BUT NOT THE VALUE OF S(V), EVEN IF S IS A SET

5. NAME SCOPING, PROCEDURE LINKAGE.

(MAPPING) FOR WHICH THE ASSIGNMENT :

$$S(V) = W ;$$

IS MEANINGFUL.

6. PROGRAM STRUCTURE.

A SETLA PROGRAM CONSISTS OF PROCEDURE DEFINITIONS AND EXECUTABLE CODE, BRACKETED IN ≠BLOCKS≠ BY =DO= AND =COMPUTE= STATEMENTS, A DO=COMPUTE BLOCK MAY CONTAIN SEVERAL PROCEDURE DEFINITIONS, AS WELL AS EXECUTABLE CODE. SUCCESSIVE DO-COMPUTE BLOCKS ARE COMPILED SEQUENCIALLY. IF A BLOCK CONTAINS EXECUTABLE CODE, IT IS EXECUTED IMMEDIATELY AFTER ITS COMPILATION. AS A CONSEQUENCE, FUNCTIONS THAT ARE INVOKED BY AN EXECUTABLE CODE FRAGMENT MUST APPEAR WITHIN EARLIER DO-COMPUTE BLOCKS THAN THEIR INVOCATION.

SIMPLEST PROGRAM ORGANIZATION WILL HAVE THE MAIN PROGRAM IN THE LAST DO=COMPUTE BLOCK, SEGREGATED FROM ALL PROCEDURE DEFINITIONS.

THE DO-COMPUTE BLOCKS PROVIDE NO NAME ISOLATION , AND SERVE NO OTHER PURPOSE THAN THIS SEGMENTATION AND REPEATED COMPILER INVOCATION. TO AVOID MEMORY OVERFLOWS, IT IS RECOMMENDED THAT DO-COMPUTE BLOCKS CONTAIN NO MORE THAN 200 LINES OF CODE.

7. USE OF THE SYSTEM

THE SYSTEM CONSISTS OF THREE PHASES :

1. A FRONT-END WHICH PARSES THE SETLA SOURCE, PRODUCES AN INTERMEDIATE TEXT FOR THE NEXT PHASE AND OUTPUTS SOURCE AND SYNTACTIC DIAGNOSTICS(IF ANY),

2. AN EXTENDED BALM COMPILER, CONSISTING OF TWO EXTENSIONS TO THE BALM SYSTEM :
   A, A SERIES OF TREE-WALKING ROUTINES WHICH TRANSFORM THE INTERMEDIATE TEXT PRODUCED IN PHASE 1 INTO VALID BALM PARSE-TREES,
   B, A SERIES OF BALM PROCEDURES(CORRESPONDING TO THE BALMSETL OF PRECEDING IMPLEMENTATIONS) WHICH MODIFY THE BALM CODE GENERATOR AND EXTEND THE MBALM MACHINE TO IMPLEMENT SETL SEMANTICS,

3. AN INTERPRETER FOR THE EXTENDED BALM SYSTEM AND THE CODE IT PRODUCES, TOGETHER WITH A RUN-TIME LIBRARY (SRTL) WHICH IMPLEMENTS THE SETL PRIMITIVES, THE INTERPRETER AND LIBRARY ARE DESIGNED TO WORK IN A DYNAMIC STORAGE AREA OF FIXED SIZE, FOR CONVENIENCE, THREE DIFFERENT FILES ARE PROVIDED, WITH INCREASING STORAGE SIZES,

THE MODULES JUST DESCRIBED RESIDE IN THE FOLLOWING FILES

1. FRONT-END : SETLA,
2. BALMSETL AND TREEWALKING ROUTINES : SAVSETL,
3. INTERPRETER AND SRTL : THREE FILES WITH THE FOLLOWING DYNAMIC STORAGE AREAS :

FILE	DYNAMIC STORAGE	RFL FOR EXECUTION
SETLA1	27000	207000 (OCTAL)
SETLA2	32000	221000
SETLA3	48000	265000

THE FOLLOWING CONTROL CARD SEQUENCE IS REQUIRED TO RUN A SETLA PROGRAM USING CIMS KRONOS 2.1 :

```
JOBNAME,TXXX,     YOUR NAME
CHARGE(TO,SOMEONE)
ATTACH(SETLA,LGO=SETLA1,TAPE8=SAVSETL,LTLLIB/UN=SETL)
RFL(150000)
SETLA,
RFL(207000)
LGO,
   E-O-R
YOUR SETLA SOURCE DECK,
   E-O-F
```

ONE OF THE OTHER SRTL FILES CAN REPLACE SETLA1 , IF CARE IS

TAKEN AT THE SAME TIME TO REPLACE THE RFL CARD THAT PRECEEDS LGO WITH THE ONE INDICATED IN THE TABLE ABOVE.

INCREMENTALITY.

THE INCREMENTALITY OF BALM IS PRESERVED IN THE SETLA SYSTEM, THE STATE OF THE PROCESSOR (HEAP, STACK, SYMBOL TABLE AND POINTERS) CAN BE SAVED AT ANY POINT DURING EXECUTION, BY USING THE PROCEDURE =SAVESETL=. EXECUTION OF THIS PROCEDURE, INVOKED BY THE STATEMENT SAVESETL ; CREATES A PRIVATE SAVEFILE WHICH CAN BE USED TO RESUME EXECUTION AT SOME LATER TIME, THIS SAVEFILE WILL CONTAIN THE BALMSETL SYSTEM, PLUS ALL PROCEDURES COMPILED BY THE USER, THESE SAVEFILES ARE USABLE ON ANY ONE OF THE SETLA FILES, SO THAT IT IS POSSIBLE TO CREATE A SERIES OF PROCEDURES USING A SMALL HEAP, AND EXECUTE THEM USING A LARGER ONE, WHEN COMPILING LARGE PROGRAMS, THIS APPROACH WILL PROVIDE FASTER TURN-AROUND TIME (WHICH GIVEN THE SIZE OF THE SYSTEM, IS NOT AN ACADEMIC CONSIDERATION.)

USER SAVEFILES ARE WRITTEN ONTO TAPE9, WHICH IS RESERVED FOR THAT PURPOSE ALONE, (SEE SECTION ON INPUT-OUTPUT), TO MAKE A SAVEFILE PERMANENT UNDER THE NAME =NEWSAVE= , INSERT THE FOLLOWING CARD BEFORE THE LGO, CARD :

```
DEFINE(TAPE9=NEWSAVE)
```

TO RESUME EXECUTION FROM THAT SAVEFILE, MODIFY THE ATTACH CARD TO READ :

```
ATTACH(---  TAPE8=NEWSAVE, ---  )
```

TAPE8 IS RESERVED FOR THAT PURPOSE AND SHOULD NOT BE USED FOR OTHER I-O OPERATIONS.

CONTROL CARD PARAMETERS

THE SETLA FRONT END PROVIDES SEVERAL OPTIONS WHICH THE USER MAY SELECT BY SUPPLYING A LIST OF THE NECESSARY KEYWORDS ON THE CONTROL CARD FOR SETLA. THE KEYWORDS AND THEIR INTERPRETATION ARE AS FOLLOWS

-XRF-,THE CROSS-REFERENCE OPTION. IF THIS OPTION IS SELECTED THEN THE OUTPUT FILE WILL INCLUDE A COMPLETE CROSS REFERENCE MAP FOR ALL NAMES IN THE SETLA INPUT PROGRAM.

-SL-,CODE LIST OPTION. IF THIS OPTION IS SELECTED THEN THE SETL SOURCE PROGRAM IS LISTED ON THE OUTPUT FILE. THE DEFAULT SETTING IS -ON-.

-HELP- REQUEST DEBUGGING AIDS. DEFAULT IS OFF. IF DEBUGGING AIDS ARE REQUESTED, THEN THE FRONT-END WILL INSERT CALLS TO TRACE ROUTINES IN THE BALM SYSTEM AT KEY POINTS OF THE USERS SOURCE CODE. DETAILS OF THE USE OF THESE FEATURES ARE CONTAINED IN THE SECTION ON -DEBUGGING AIDS-

-SN- REQUESTS STATEMENT-BY-STATEMENT TRACE.THIS DEBUGGING AID CAN BE ACTIVATED WITHOUT THE FULL -HELP- FEATURE (WHICH PROVIDES SEVERAL ADDITIONAL TRACING PROCEDURES). DEFAULT IS OFF.

-ABT- ABORT ON LEXICAL ERRORS,DEFAULT IS -ON-,IF LEXICAL ERRORS ARE DETECTED IN THE SOURCE, EXECUTION WILL BE TERMINATED. IF THIS OPTION IS DISABLED,BALMSETL COMPILATION AND EXECUTION WILL PROCEED AS FAR AS POSSIBLE.

THESE OPTIONS ARE SPECIFIED BY PROVIDING A LIST OF NECESSARY KEYWORDS AND VALUES,ENCLOSED IN PARENTHESES, ON THE CONTROL CARD WHICH INITIATES EXECUTION OF THE SETLA FRONT-END. A KEYWORD IS ASSIGNED A VALUE BY FOLLOWING THE KEYWORD WITH AN EQUALS SIGN (=) AND THE VALUE, THE VALUE MUST BE A NON-NEGATIVE INTEGER, ONE OF THE WORDS -ON-, -YES- OR -T- (WHICH CORRESPOND TO VALUE OF 1), OR ONE OF THE WORDS -OFF- -NO- OR -F- (WHICH CORRESPOND TO VALUE OF 0).
ALL OF THE FOLLOWING CONTROL CARDS ARE EQUIVALENT:

```
SETLA, (HELP)

SETLA,  (HELP,SL=1)

SETLA,  (HELP=YES,SL,ABT=ON)
```

HERE ARE SOME EXAMPLES OF PARAMETER LISTS:

```
SETLA,
       LIST INPUT,NO TRACING,ABORT IF LEXICAL ERRORS FOUND.

SETLA,  (ABT=0,HELP)
       LIST INPUT, ENABLE TRACING FEATURES, EXECUTE EVEN IF
       LEXICAL ERRORS PRESENT.

SETLA,  (SN=1,SL=0)
       NO INPUT LISTING, ENABLE STATEMENT-BY-STATEMENT TRACE.
```

DEBUGGING FEATURES OF SETLA

THE SETLA TRANSLATOR PROVIDES SEVERAL USEFUL DEBUGGING FEATURES, AT THE USERS REQUEST, THE TRANSLATOR WILL INSERT IN THE BALM CODE CALLS TO SYSTEM TRACE ROUTINES WITHIN THE BALMSETL SYSTEM, AT EXECUTION TIME THE VALUES OF GLOBAL VARIABLES (SWITCHES) MAY BE SET BY THE USER TO CONTROL THE GENERATION OF DEBUG OUTPUT BY THESE TRACE ROUTINES, THE TRACE FEATURES CURRENTLY AVAILABLE PROVIDE FOR THE TRACING OF PROGRAM ENTRY AND RETURN TO SUBPROGRAMS, TRACE OF ASSIGNMENTS STATEMENTS, AND STATEMENT-BY-STATEMENT EXECUTION TRACE.

AS AN EXAMPLE OF HOW THE TRACE PACKAGE WORKS CONSIDER THE SETLA SEQUENCE FOR THE LAST FEW LINES IN PROCEDURE -P-

```
,,, A=10; RETURN (A); END P;
```

THIS TRANSLATES INTO THE EQUIVALENT OF THE BALM SEQUENCE

```
A = 10, RETURN A, END P;
```

IF ENTRY/EXIT TRACING IS REQUESTED THEN THE CODE GENERATED IS

```
A=10,
ATEXVAL=A,
ATEXITV(3,=P,ATEXVAL),
RETURN (ATEXVAL),
```

IF STORES TO -A- ARE BEING TRACED, THEN THE CODE IS

```
A=10,
ATSETV(2,=P, =A, A),
ATEXVAL=A,
ATEXITV(3,=P,ATEXVAL),
RETURN (ATEXVAL),
```

IF THE OUTPUT IS ONLY TO INCLUDE THE TRACES OF ASSIGMENTS AND ENTRY-EXIT STATEMENTS, THE OUTPUT WILL BE AS FOLLOWS

```
  ***** AT LINE 2 IN P A IS 10
  ***** RETURN FROM 3 IN P WITH VALUE 10
```

IF IN ADDITION, THE STATEMENT-BY-STATEMENT TRACE IS ACTIVATED, THE ADDITIONAL CALL

```
     ATSN(2,=P)
```

IS INSERTED IN THE CODE, AND THE FOLLOWING LINE WILL APPEAR IN THE OUTPUT :

```
  ***** LINE 2 IN P
```

THE EXAMPLE ILLUSTRATES THE THREE LEVELS OF USER CONTROL OF THE DEBUG OPTIONS

A, WHETHER TO GENERATE CALLS TO TRACE ROUTINES
B, WHICH KINDS OF CALLS TO GENERATE
C, EXECUTION TIME CONTROL OVER OUTPUT BY CHANGING VALUES OF GLOBAL VARIABLES USED BY THE TRACE ROUTINES.

WE NOW DISCUSS EACH OF THESE OPTIONS IN MORE DETAIL.

ACTIVATING DEBUG PACKAGE IN SETLA TRANSLATOR

THE "HELP" OPTION ON THE SETLA CONTROL CARD ACTIVATES ALL TRACING PROCEDURES : ENTRY/EXIT TRACE, ASSIGNMENT TRACE, AND STATEMENT-BY-STATEMENT EXECUTION TRACE.

THIS LAST TRACE CAN BE ENABLED INDEPENDENTLY BY THE "SN" OPTION ON THE SETLA CONTROL CARD.

IF ONE OR BOTH OF THESE OPTIONS ARE PRESENT, THE APPROPRIATE CALLS ARE INSERTED IN THE CODE, HOWEVER, THE USER CAN CONTROL THE EXECUTION OF THESE CALLS BY MEANS OF THE FOLLOWING GLOBAL FLAGS:

```
ATEXTRC :     CONTROLS ENTRY/EXIT TRACING.
ATEGTRC :     CONTROLS ASSIGNMENT TRACING.
ATSNTRC :     CONTROLS STATEMENT-BY-STATEMENT TRACING.
```

THE DEFAULT SETTINGS ARE TRUE, TRUE, FALSE RESPECTIVELY.

IF THE TRACES HAVE BEEN ENABLED, BUT THE CORRESPONDING FLAGS ARE OFF, THE CALLS TO TRACING ROUTINES ARE STILL EXECUTED, BUT NO OUTPUT WILL BE PRODUCED.

WITH THE DEFAULT SETTINGS SPECIFIED ABOVE, A SIMPLE INVOCATION OF "HELP" ON THE SETLA CONTROL CARD WILL RESULT IN THE LISTING OF ENTRY/EXIT AND ASSIGNMENT TRACES; IN ORDER TO OBTAIN THE STATEMENT TRACE, THE USER CAN INSERT IN HIS PROGRAM THE STATEMENT :

```
ATSNTRC = T, ;
```

SIMILAR ASSIGNMENTS TO THE OTHER TRACE FLAGS CAN BE USED SELECTIVELY TO TRACE ONLY CERTAIN PORTIONS OF A PROGRAM.

NOTE THAT IF THE HELP OPTION IS SPECIFIED, TRACING CALLS ARE INSERTED TRHOUGHOUT THE CODE, EVEN IF THE TRACING OUTPUT WILL LATER BE TRIMMED BY JUDICIOUS USE OF THE TRACE FLAG. THESE TRACING CALLS EXPAND THE RESULTING CODE NOTICEABLY, AND MAY REQUIRE FOR ITS COMPILATION AND EXECUTION A LARGER HEAP THAN THE ORIGINAL UNTRACED PROGRAM .

TO RESTRICT THE INSERTION OF ENTRY/EXIT AND ASSIGNMENT TRACES , THE -CHECK- STATEMENT HAS BEEN PROVIDED. IT IS DESCRIBED IN THE FOLLOWING SECTION.

SETLA STATEMENTS CONTROLLING DEBUG FEATURES

A ≡CHECK≡ STATEMENT HAS BEEN ADDED TO THE SETLA LANGUAGE. THIS STATEMENT HAS THE FORM

```
<CHECK / NOCHECK> < STORES / ENTRY >   ;
```

WHERE #/# INDICATES THAT ONE OF THE OPTIONS IS ALLOWED.

IF THE STATEMENT :

```
CHECK ENTRY ;
```

APPEARS IN THE USERS SOURCE CODE, THEN SUBSEQUENT PROCEDURES ARE COMPILED WITH ENTRY/EXIT CALLS INSERTED, AND THE CORRESPONDING FLAG, ATEXTRC , IS SET ON. IF THE STATEMENT

```
NOCHECK ENTRY ;
```

IS ENCOUNTERED LATER ON, SUBSEQUENT PROCEDURES ARE COMPILED WITHOUT TRACING CALLS.

THE SAME APPLIES, PARI PASU, TO THE STATEMENT:

```
CHECK STORES;
```

AND FOLLOWING ASIGNMENT STATEMENTS.

FOR REFERENCE, WE LIST HERE THE TRACE ROUTINES CALLED BY THE DEBUGGING AIDS PACKAGE :

ATSN(LINE, SUB) - CALLED AT END OF EXECUTABLE STATEMENTS. THIS PROCEDURE SAVES ITS ARGS, AND MAINTAINS A LIST OF THE LAST 10 STATEMENTS EXECUTED. THIS LIST IS DISPLAYED AFTER A USER CRASH.

ATENTRY(SUB) - CALLED WHEN ENTER ROUTINE
PRINTS ARGUMENT IF -ATEXTRC- HAS VALUE -TRUE-

ATEXIT(LINE,SUB) - CALLED WHEN RETURN FROM ROUTINE
PRINTS ARGS IF -ATEXTRC- IS -TRUE-

ATSETV(LINE,SUB,NAME,VAR) - CALLED BY SIMPLE ASSIGNMENT
PRINTS ARGUMENTS IF -ATEQTRC- IS -TRUE-

ATSETLSN(LINE,SUB, NAMELIST, VARLIST)-CALLED FOR MULTI-ASSIGN
EG <A,B,C> = S;
PRINTS ARGUMENTS IF -ATEXTRC- IS TRUE

WHERE -LINE- IS INTEGER GIVING STATEMENT NUMBER, AND -SUB- IS SUBPROGRAM NAME, -NAME- IS NAMEOF VARIABLE (PRECEDED BY =), -VAR- IS VARIABLE NAME, -NAMELIST- IS LIST OF QUOTED NAMES, AND -VARLIST- IS LIST OF VALUES.

8. FORMAL GRAMMAR

THE FOLLOWING LL1 BACKUS GRAMMAR FOR SETLA MAY AID IN THE UNDERSTANDING OF VARIOUS OF ITS SYNTACTIC DETAILS. THE NOTATIONS USED IN THE GRAMMAR ARE AS FOLLOWS:

```
<STYPE>              =DENOTES A SYNTACTIC TYPE
<*LTYPE>             =DENOTES A LEXICAL TYPE
LITERAL,≠LITERAL≠    =DENOTE LITERALS
<-COMMENT>           =DENOTES A COMMENT
<STYPE*>             =DENOTES INDEFINITELY MANY REPETITIONS
                          OF A SYNTACTIC TYPE
<STYPE(M,N)>         =DENOTES A LIMITED NUMBER OF REPETITIONS
                          OF A SYNTACTIC TYPE
                      M:MINIMUM NUMBER REQUIRED
                      N:MAXIMUM NUMBER ALLOWED
```

SUCCESSIVE ALTERNATIVE EXPANSIONS OF A SYNTACTIC TYPE ARE INDICATED BY SUCESSIVE EQUALITY SIGNS, AS FOLLOWS:

```
<STYPE>        =   (FIRST ALTERNATIVE EXPANSION)
               =   (SECOND ALTERNATIVE EXPANSION)
               =   (ETC.,ETC.)
```

LEXICAL TYPES:

THE FOLLOWING LEXICAL TYPES OCCUR IN SETLA AND APPEAR IN THE FORMAL GRAMMAR GIVEN BELOW:

```
<*NAME>        =A VARIABLE,SUBROUTINE,OR FUNCTION NAME, NOT
                    PERIOD-TERMINATED
               EXAMPLES: X,Y,Z0,Z000,Z0A1
<*OPNAME>      =THE NAME OF A SYSTEM OR USER-DEFINED
                    OPERATOR, POSSIBLY PERIOD-TERMINATED
               EXAMPLES:   MAX., MIN., +, //, MYMONAD., MYDYAD.,
<*QNAME>       =ONE OF A GROUP OF PERIOD-TERMINATED NAMES KNOWN
                    TO THE COMPILER AS THE NAME OF A SPECIAL
                     QUANTITY OR QUANTITY-RETUNING, Q-PARAMETER
               EXAMPLES:  T.,F.,NL.,NULC.,NULT.,NEWAT.
<*LOGOP>       =ONE OF A SPECIAL GROUP OF OPERATOR NAMES KNOWN
                    TO THE COMPILER AS LOGICAL OPERATORS OF
                                        HIGH PRIORITY
               EXAMPLES: GT.,EQ.,NE.,GE.,INCS.
<*CONST>       AN INTEGER OR STRING CONSTANT
               EXAMPLES:  17,960,≠THE LONG GOODBYE≠
```

```
<PROGRAM>       =<STATEMENT> <STATEMENT*>

<BLOCK>         =<STATEMENT> <STATEMENT*>

<LABEL>         = <*NAME> :

<STATEMENT>     = <LABEL> <STATEMENT>
                =IF <EXPN> THEN <BLOCK> <ELSEIF*> ELSE <BLOCK>
                                                    <ENDER>
                =IF <EXPN> THEN <BLOCK> <ELSEIF*> <ENDER>
                =(∪ <ITERATOR>) <BLOCK> <ENDER>
                =(WHILE <EXPN> DOING <BLOCK>) <BLOCK> <ENDER>
                =(WHILE <EXPN>) <BLOCK> <ENDER>


                =<*NAME> (<EXPN> <COMEXPN*>);
                = <*NAME> ;
                =<"SUBROUTINE CALL IN INFIX FORM!>
                     <EXPN> <*OPNAME> <EXPN> ;
                = <*OPNAME> <EXPN> ;
                = <*OPNAME> ;
                =<"CALL OF SUBROUTINE TO ALL ELEMENTS OF SET:>
                     <*NAME> [ <EXPN> <COMEXPN*> ] ;
                = <*OPNAME> [ <EXPN> ] ;


                = ASSERT <EXPN>;


                = <*NAME> ≠=≠ <EXPN> ;
                = ≠HD,≠ <*NAME> ≠=≠ <EXPN> ;
                = ≠TL,≠ <*NAME> ≠=≠ <EXPN> ;

                = <"     INDEXED ASSIGNMENT FORMS!>
                     <*NAME> (<EXPN> <COMEXPN*>) ≠=≠ <EXPN> ;
                =    <*NAME> ≤<EXPN> <COMEXPN*>≥ ≠=≠ <EXPN> ;
                =    <*NAME> [<EXPN> <COMEXPN*>] ≠=≠ <EXPN> ;
                = <*NAME> ( <EXPN> : <EXPN> ) = <EXPN> ;


                = <CHECKWORD> <CHECKOP>;


                =GOTO <EXPN>;
                =GO TO <EXPN>;
                =NOOP;
                =COMPUTE;
                =DEFINE <DEFORM> <BLOCK> <ENDER>
                =DEFINEF <DEFORM> <BLOCK> <ENDER>
                =LOCAL <NAMELIST> ;
```

```
                = ≠PRINT,≠ <EXPN> <COMEXPN*> ;
                = ≠READ,≠  <NAMELIST> ;
                = ≠WRITE≠ <NAMELIST>;


                =FINISH ;
                = ≠<≠ <NAMELIST> ≠>≠ ≠=≠ <EXPN> ;
                = RETURN ;
                = RETURN  <EXPN> ;
                =<- SPECIAL ≠RETURN≠ STATEMENT FOR USE IN
                          CONNECTION WITH CODE BLOCKS;>
                     RETN  <EXPN>  ;
                =DO;

<DEFORM>        = <*NAME> ( <NAMELIST> )
                = <*NAME> <*OPNAME> <*NAME>
                = <*NAME>
                = <*OPNAME> <*NAME>
                = <*OPNAME>


<ELSEIF>        = ELSE IF  <EXPN> THEN <BLOCK>

<ENDER>         = ;
                = END ;
                = END <NOSEMS(1,5)> ;
<NOSEMS>        = <*NOSEMI>


<ITERATOR>      = <ITEREXPN> <COMITEREXPN*> ↑ <EXPN>
                = <ITEREXPN> <COMITEREXPN*>
<COMITEREXPN>   = , <ITEREXPN>
<ITEREXPN>      = <*NAME> → <EXPN>
                = <EXPN> <COMPAREOP> <*NAME> <COMPAREOP>

<COMPAREOP>     = >
                => ≠=≠
                = <
                = < ≠= ≠
<COMEXPN>       = , <EXPN>


<NAMELIST>      = <*NAME> <NAMEC>
<NAMEC>         = , <*NAME>


<CHECKWORD>     =  CHECK
```

```
                = NOCHECK
<CHECKOP>       = STORES
                = TIME
                = ENTRY

<EXPN>          = <FACTOR> <*OPNAME> <EXPN>
                = <FACTOR>
                = IF <EXPN> THEN <EXPN> <ELSEXPN*> ELSE <EXPN>

<ELSEXPN>       = ELSE IF <EXPN>

<FACTOR>        = <*OPNAME> <FACTOR>
                = Ξ <ITEREXPN> <COMITEREXPN> ↑ <FACTOR>
                = ν <ITEREXPN> <COMITEREXPN> ↑ <FACTOR>
                = [ <*OPNAME> : <ITERATOR> ] <FACTOR>
                = <ELEMENT> <*LOGOP> <FACTOR>
                = <ELEMENT>

<ELEMENT>       = <ATOM> ( <EXPN> : <EXPN> )
                = <ATOM> ( <EXPN> <COMEXPN*> )
                = <ATOM> ≤ <EXPN> <COMEXPN*> ≥
                = <ATOM> [ <EXPN> <COMEXPN*> ]
                = <ATOM>

<ATOM>          = <*NAME> ( <EXPN> <COMEXPN*> )
                = <*NAME> ≤ <EXPN> <COMEXPN*> ≥
                = <*NAME> [ <EXPN> <COMEXPN*> ]
                = <*NAME>
                = <*CODE BLOCK TREATED AS EXPRESSION>
                    [: <BLOCK> ]
                = ≤ <*NAME> → <EXPN> ,<ITEREXPN> ↑ <EXPN> ≥
                = ≤ <*NAME> → <EXPN> ,<ITEREXPN>  ≥
                = ≤ <EXPN> , <ITERATOR> ≥
                = ( <EXPN> )
                = ( <*OPNAME>)
                = ≤:≥
                = ≤: <EXPN> <COMEXPN*> ≥
                = ≠<≠ <EXPN> <COMEXPN*> ≠>≠
                = <*QNAME>
                = <*CONST>


END-OF-FORMAL-GRAMMAR
```

A. IMPORTANT NOTES:

1. IT IS FORBIDDEN TO MODIFY THE VALUE OF THE ITERATION VARIABLE OR OF THE ITERATION RANGE WITHIN THE BODY OF A FORALL ITERATION.
THE FOLLOWING IS THUS ILLEGAL:

```
FORALL X EL S REPEAT DO
   .....
   S=S LESS X
   .....
END
```

2. ERRORS MAY RESULT IF A RECURSIVE PROCEDURE CALLS ITSELF FROM WITHIN A LOOP OVER A SET, FOR EXAMPLE,

```
DEFINEF REC(S );
/*SUMS NESTED TUPLES OF INTEGERS*/
LOCAL X,VAL;
VAL=0;
IF (TYPE, S) ,EQ. INT, THEN RETURN S;;
(∀X -> S) VAL=VAL+REC(X);
RETURN VAL;
END;
```

MAY CAUSE PROBLEMS. IF CODE OF THIS SORT IS NECESSARY, THEN THE USER MUST SUPPLY A TEMPORARY VARIABLE, SAY "XT", IN THE LOCAL BLOCK, AND EXPRESS THE SET ITERATION AS A "WHILE" LOOP, AS FOLLOWS,

```
DEFINEF REC(S );
/*SUMS NESTED TUPLES OF INTEGERS*/
LOCAL X,VAL, XT;
VAL=0;
IF (TYPE, S) ,EQ. INT, THEN RETURN S;;
XT = NILVECT; $INITIALISE FOR ITERATION
X=NEXTELT(S,XT); $SET X
(WHILE X NE, OM, DOING X=NEXTELT(S,XT);)
VAL=VAL+REC(X);
END WHILE;
RETURN VAL;
END;
```

IN THE CODE ABOVE, TWO BALMSETL RESERVED WORDS HAVE BEEN UTILIZED:

1. "NILVECT" IS A SYSTEM CONSTANT WHICH SERVES AS A FLAG TO INDICATE THE BEGINNING OF AN ITERATION.
2. "NEXTELT" IS A SYSTEM PROCEDURE WHICH ACTUALLY PERFORMS THE ITERATION.

THE APPEARANCE OF THESE KEYWORDS (WHICH SHOULD NOT BE USED FOR ANY OTHER PURPOSE IN A SETLA PROGRAM) IS A BLATANT (UNESTHETIC) PATCH. IT IS IMPOSED BY THE BASIC IMCOMPATIBILITIES BETWEEN BALM AND SETL NAMESCOPING RULES.

THE WHILE LOOP CORRESPONDS TO THE CODE THAT WOULD BE GENERATED IF A "FORALL" LOOP WERE WRITTEN, HOWEVER, THE REQUIRED TEMPORARY "XT" IS NO LONGER GLOBAL, AND IS THUS RESTORED CORRECTLY IN THE EVENT OF RECURSIVE CALLS.

3. COPYING AND DIRECT MODIFICATION OF DATA OBJECTS IN SETLA:

THE USER SHOULD BEWARE OF ERRORS WHICH MAY BE CAUSED BY THE FOLLOWING LOGICAL DISCREPANCY BETWEEN SETL AND BALM. SETL IS A CONSISTENTLY "VALUE ORIENTED" LANGUAGE, IN WHICH, AS A MATTER OF LOGICAL PRINCIPLE, OPERATIONS WHICH MODIFY EXISTING VARIABLE VALUES CREATE ENTIRELY NEW DATA STRUCTURES, AND LEAVE ALL OTHER VARIABLE VALUES UNCHANGED. BALM ON THE OTHER HAND IS "ADDRESS AND POINTER" ORIENTED IN ITS TREATMENT OF COMPOUND (THOUGH NOT OF SIMPLE) DATA OBJECTS, SO THAT WHEN A COMPOUND OBJECT A IS MADE PART OF ANOTHER SUCH OBJECT B, THE VALUE OF B MAY SUBSEQUENTLY CHANGE WHEN A IS MODIFIED. TO SUPPRESS SUCH EFFECTS, WHICH ARE NOT CONSONANT WITH THE PURE INTENT OF SETL, IT MAY BE NECESSARY TO INSERT OCCASIONAL CALLS TO THE "CREATE INDEPENDENT NEW COPY" FUNCTION COPY(X).

THE CURRENT IMPLEMENTATION INSERTS AUTOMATIC COPIES IN MOST SITUATIONS WHERE A COMPOSITE OBJECT IS BEING MODIFIED, OR RETRIEVED FROM A LARGER OBJECT, FOR EXAMPLE ,IN THE ASSIGNMENT

```
Y = F(X) ;
```

THE VALUE ASSIGNED TO Y IS A COPY OF THE VALUE RETRIEVED BY THE FUNCTIONAL APPLICATION, SO THAT SUBSEQUENT MODIFICATIONS OF Y WILL NOT HAVE SIDE-EFFECTS ON THE SET F. HOWEVER, THE SIMPLE ASSIGNMENT Y = X ; DOES NOT PRODUCE A COPY OF X, SO THAT MODIFICATIONS TO ONE OF THE VARIABLES MIGHT PROPAGATE TO THE OTHER, EXPERIENCE WITH SETL SO FAR SEEMS TO INDICATE THAT ONLY RARELY DOES THIS FORCE THE USER TO INSERT EXPLICIT COPIES IN HIS PROGRAM. HOWEVER, IT MIGHT BE WISE TO KEEP THIS IRREGULARITY IN MIND WHEN DEBUGGING.

6. INPUT-OUTPUT

THE SYSTEM PROVIDES THE FILES TAPE1,TAPE2,...,TAPE7 FOR USER I=O OPERATIONS, TAPE1 AND TAPE2 ARE THE KRONOS (OR SCOPE) INPUT AND OUTPUT FILES RESPECTIVELY.

IN ADDITION, BALMSETL DEFINES INTERNALLY A BUFFER CALLED -INFILE- , WHICH CAN BE ASSOCIATED WITH ONE OF THE SYSTEM FILES, AND WHICH SPECIFIES THE FILE FROM WHICH DATA IS TO BE READ. INFILE IS EQUIVALENCED TO -INPUT- BY DEFAULT.

THE FORMAT OF THE I/O STATEMENTS IS AS FOLLOWS:

PRINT,O1,O2,...ON;	PRINTS SETL OBJECTS O1,..ON IN SETL EXTERNAL FORM(SEE BELOW) ON FILE OUTPUT, THE ITEMS ARE SEPARATED BY ONE BLANK.
WRITE,N, O1,O2,...,ON ;	SIMILAR TO PRINT, BUT DATA IS WRITTEN ON FILE TAPEN, N MUST BE INTEGER<8.
READ, O1,O2,...,ON ;	READS THE SETL EXTERNAL FORM OF N OBJS FROM FILE ≠INFILE≠, AND STORES THE CORRESPONDING VALUES IN THE VARIABLES O1,..,ON. ITEM DELIMITER IS A BLANK OR A SLASH. IT IS RECOMMENDED THAT THE SLASH BE USED TO MARK THE END OF COMPLICATED SETS OR TUPLES.

-INFILE- CAN BE REDEFINED BY MEANS OF THE -MAKFILE- PROCEDURE. THE STATEMENT :

```
INFILE=MAKFILE(N,SIZ) ;
```

WHERE N AND SIZ ARE INTEGERS, SPECIFIES THAT THE NEXT READ OPERATION IS TO TAKE PLACE ON FILE TAPEN, WHERE DATA IS WRITTEN -SIZ- CHARACTERS PER RECORD. ON TAPE1 (THE STANDARD INPUT FILE) THIS PARAMETER HAS A DEFAULT SETTING OF 80.

THE OUTPUT LINE SIZE CAN BE SPECIFIED BY THE SAME MECHANISM, FOR EXAMPLE. THE STATEMENT :

```
DUMMY= MAKFILE(2,72) ;
```

WILL FORCE THE OUTPUT ON TAPE2 TO BE PRINTED IN 72 COLUMNS.

THE DEFAULT SETTINGS FOR THIS PARAMETER ARE AS FOLLOWS:

TAPE2 (STANDARD OUTPUT FILE) : 130 CHARS/LINE
ALL OTHER FILES : 80 CHARS/LINE.

EXTERNAL FORM OF OBJECTS (EXAMPLES)

INTEGERS	1 23 -5 7186314159265	
REALS	0.1 10.2E-6	
BIT STRINGS	11B 77770 10B535353	
CHARACTER STRINGS	≠ABCDEFGHIJK≠	
BLANK ATOMS	BLK123	NOT READABLE
LABELS	LAB.45	NOT READABLE
PROCEDURES	FUN. RANDOM	NOT READABLE
UNDEFINED VALUE	OM.	
EMPTY SET	NL.	
EMPTY TUPLE	NULT.	
SET	≤1 2 3 4≥	
TUPLE	<1 2 3 4>	
TRUE, FALSE	T. F.	

C. BOOLEAN OPERATORS.

THE BOOLEAN EXPRESSION :

A AND. B

IS EQUIVALENT TO (AND EVALUATED AS)

(IF A THEN B ELSE F.)

I.E. IF A IS FALSE, THEN B WILL NOT BE EVALUATED.
SIMILARLY, THE BOOLEAN EXPRESSION

A OR. B

IS EQUIVALENT TO

(IF A THEN T. ELSE B)

NOTE THAT THE ARGUMENT OF AN -IF- STATEMENT WILL BE TAKEN TO BE TRUE IF IT IS NOT THE BOOLEAN F. (OR 0B), THE INTEGER 0, THE UNDEFINED VALUE OM., THE NULL SET NL., WILL ALL ACT AS THE BOOLEAN T. IN THIS CASE.

D. MISCELLANEOUS BALMSETL FUNCTIONS OF INTEREST

LEVEL

THIS IS AN INTEGER VARIABLE (WHOSE INITIAL VALUE IS 10) WHICH FIXES THE DEPTH OF THE PROCEDURE CALL NESTING DISPLAYED WHEN A TERMINAL ERROR IS DETECTED. THIS DEPTH CAN BE CHANGED BY A REGULAR ASSIGNMENT SUCH AS

```
LEVEL=20;
```

CRASHMAX

THIS INTEGER VARIABLE SPECIFIES THE MAXIMUM NUMBER OF CRASHES ALLOWED BEFORE TERMINATION OF A PROGRAM. IT IS INITIALIZED TO 5.

STRINGOF(O)

THIS FUNCTION RETURNS A CHARACTER STRING WHICH IS THE EXTERNAL REPRESENTATION OF ATOMIC OBJECT O.

E. RESERVED WORDS

THE FOLLOWING IS AN ALPHABETICAL LIST OF THE RESERVED WORDS OF THE BALMSETL SYSTEM. THESE IDENTIFIERS SHOULD NOT BE USED AS NAMES OF USER CREATED VARIABLES.

AUGMENT
BOF
BOFN
CODEQ
DIMINISH
DIMF
DIMFN
DO
ELSE
ELSEIF
END
EQ
EQUAL
FALSE
FOR
GARBCOLL
GE
GENSET
GENTUP
GO
GOTO
GT
HD
HEAD
IF
IN
INCS
INDEX
INEG
INPUT
IS
LAND
LE
LESF
LESFN
LESS
LNOT
LT
MAKFILE
MAX
MIN
NE
NEQUAL
NEWAT
NIL
NILQ
NILVECT
NL
NOT
NPOW
NULB
NULC
NULLSET
NULT
OCT
OR
POW
PRINT
PROC
QUOTE
READ
RETURN
SAVESETL
SHIFT
SOF
SOFN
SSOF
SSOFN
TAIL
TAIKATIV
THEN
TIME
TRUE
TYPE
VECTOR
WHILE
XOR

NEXTELT (SEE SEC.8A.)

10, SAMPLE PROGRAMS IN SETLA

8,SOME SAMPLE PROGRAMS
THE REMAINDER OF THIS MANUAL CONTAINS SEVERAL PROGRAMS WRITTEN IN SETLA, THE SOURCE FOR THESE PROGRAMS IS AVAILABLE ON THE PERMANENT FILE =STESTPL=, WHICH IS AN UPDATE OLDPL.

```
/* THESE PROGRAMS ARE PRESENTED TO ILLUSTRATE THE USE OF THE
     SETLA LANGUAGE.  IN PARTICULAR, DIFFERENCES IN USAGE BETWEEN
     SETL AND SETLA ARE INDICATED.ONLY SKETCHY INDICATIONS
     OF THE UNDERLYING STRATEGY USED IN THESE ALGORITHMS IS GIVEN
     IN THE TEXT WHICH FOLLOWS.  FOR ADDITIONAL EXPLANATION
     SEE THE SECTIONS OF THE ≠ ON PROGRAMMING ≠ CITED IN
     CONNECTION WITH THE PROGRAMS GIVEN BELOW          */
```

POCKET-SORT SORTING ALGORITHM: (O'P. VOL II.P.64)

```
/* THIS ROUTINE SORTS BY THE ≠DISTRIBUTION AND COLLECTION≠
    METHOD USED ON MECHANICAL CARD SORTERS */

DO;
DEFINEF POCKSORT(SEQ,P);
/*SEQ IS A SEQUENCE OF INTEGERS TO BE SORTED,
    P IS THE NUMBER OF POCKETS TO BE USED,*/
/*  MULTI TRUE AS LONG AS MORE THAN ONE POCKET HAS CARDS          *
MULTI = T,; Q= 1;
(WHILE MULTI DOING Q=Q*P;)
/* THE ITEMS ARE DISTRIBUTED INTO POCKETS BY KEYS
    INCREASING AS THEIR RESIDUES MODULO P,P**2, P**3, ETC.,
    AND THEN REGATHERED IN THE SEQUENTIAL ORDER OF THE
    POCKETS  */
    SEQ= GATHER(P,DIST(SEQ,P,Q));
END WHILE;
RETURN SEQ;
END POCKSORT;

DEFINEF DIST(SEQ,P,Q);
        /*DISTRIBUTES SEQ AMONG P POCKETS ACCORDING TO RESIDUE
        MODULO P*Q,
        ALSO CALCULATING FLAG =MULTI=*/
POCKET=NL,;
(∀1<=K<=#SEQ)
    KEY=(SEQ(K)/Q)//P;
    POCKET(KEY,# POCKET≤KEY≥ +1)= SEQ(K);
    END ∀;
MULTI= ∃ 1<=K<P|(POCKET≤K≥ NE, NLL,);
RETURN POCKET;
END DIST;

DEFINEF GATHER(P,POCKET);
/*  GATHERS DISTRIBUTED ITEMS IN SEQUENCE OF POCKETS    */

RETURN [+:0<=K<P, 1<=J<= #POCKET≤K≥] <POCKET(K,J)>;
END GATHER;
COMPUTE;
```

```
     FORD-JOHNSON TOURNAMENT SORT      (O,P, VOL II, PAGE 66)
 /* THIS IS THE FORD-JOHNSON ≠MINIMUM COMPARISON SORTING
     METHOD.  SEE THE CITED REF. */

/*                                PLACE                                  */

DO;
DEFINEF PLACE(ELT,NELTS,BIGR);
/* THIS AUXILIARY ROUTINE USES A BINARY SEARCH PROCDCEDURE
     TO DETERMINE THE PROPER POSITION OF ≠ELT≠ WITHIN THE
     SEQUENCE ≠SEQ≠ */
LOCAL BOT,TOP,MID;
BOT=1;
TOP=NELTS;
(WHILE(TOP-BOT) GT, 1)
     MID=(TOP+BOT)/2;
    IF BIGR(SEQ(MID),ELT)THEN
         TOP=MID;
/* ≠BIGR≠ IS A BOOLEAN-VALUED COMPARISON FUNCTION      */
         ELSE BOT=MID;;
END WHILE;
IF BIGR(SEQ(BOT),ELT) THEN RETURN BOT;
     ELSE IF BIGR(SEQ(TOP),ELT) THEN RETURN TOP;
     ELSE RETURN TOP+1;;
END PLACE;
COMPUTE;


/*                               INSERT                                  */

DO;
 /*  THIS AUXILIARY FUNCTION INSERTS ≠ELT≠ AT A SPECIFIED PLACE
     WITHIN ≠SEQ≠ */
DEFINE INSERT(ELT,PLACE);
SEQ=(SEQ(1:PLACE-1)+<ELT>)+SEQ(PLACE:(+SEQ-(PLACE-1)));
RETURN;
END INSERT;
COMPUTE;
/*                                FORDJ                                  */

DO;
DEFINEF FORDJ(PAIR);
LOCAL ITEM1,ITEM2,XTRA,MAP,ITEMS2,QSEQ,JTOP,JBOT,NELTS,J,BIGR,
      ITEMS;
 /* THIS IS THE RECURSIVE ≠TOURNAMENT SORT≠ PROCEDURE PROPER,
     THE INPUT IS ASSUMED TO HAVE THE FORM  <ITEMS, BIGR>, WHERE
     ≠ITEMS≠ IS SEQUENCE OF ITEMS TO BE SORTED, AND ≠BIGR≠ IS THE
     THE BOOLEAN-VALUED FUNCTION USED TO COMPARE TWO ITEMS */
ITEMS=HD, PAIR;
```

```
BIGR=HD, TL, PAIR;
 /* IF THE SEQUENCE CONSISTS OF ONE OR TWO ITEMS,
      ITS TREATMENT IS OBVIOUS */
IF(#ITEMS)EQ, 1 THEN RETURN <ARB, ITEMS>;;
IF(#ITEMS)EQ, 2 THEN
      ITEM1=ITEMS(1);
      ITEM2=ITEMS(2);
IF BIGR(ITEM2,ITEM1) THEN RETURN <ITEM1,ITEM2>;
          ELSE RETURN <ITEM2,ITEM1>;;
END IF(#ITEMS;
/*  OTHERWISE DIVIDE THE ITEMS INTO TWO-ELEMENT SETS,
      INTRODUCING A ≠DUMMY≠ EXTRA ITEM IF NECESSARY */
XTRA=NEWAT,;
ITEMS2=NULT,;
MAP=NL,;
    IF((#ITEMS)//2)NE, 0 THEN ITEMS((#ITEMS)+1)=XTRA;;
(WHILE ITEMS NE, NULT,)
      ITEM1=ITEMS(2);
      ITEM2=ITEMS(1);
      N1=#ITEMS-2;
      ITEMS=ITEMS(3:N1);
/*  MAP THE BIGGER OF THE TWO ITEMS IN EACH PAIR INTO THE SMALLER,
      AND CREATE A HALF-WIDTH SEQUENCE CONSISTING OF THE BIGGER ITE
      OF EACH PAIR. */
IF ITEM1 EQ, XTRA THEN X=F,;
      ELSE
      X=BIGR(ITEM2,ITEM1);;
      IF X THEN <ITEM1,ITEM2,JK>=<ITEM2,ITEM1>;;
ITEMS2(#ITEMS2+1)=ITEM2;
      MAP(ITEM2)=ITEM1;
END WHILE;
/* USE THE TOURNAMENT SORT RECURSIVELY TO SORT THE
      HALF-LENGTH SEQUENCE. */
SEQ=FCRDJ(<ITEMS2,BIGR>);
/*  NOW, USING BINARY SEARCH, INSERT THE REMAINING ELEMENTS OF THE
      ORIGINAL SEQUENCE INTO THEIR PROPER POSITION. */
  OSEQ = [+:1<=N<=#SEQ]<MAP(SEQ(N))>;
(~1<=J<=(#OSEQ))
            IF OSEQ(J) NE, XTRA THEN
            NELTS=#SEQ;
            INSERT(@SEQ(J),PLACE(OSEQ(J),NELTS,BIGR));; END ~;
RETURN SEQ;
END FCRDJ;
COMPUTE;
```

```
/*                                ALPHBIGR                                */
 /* HERE IS AN ALPHABETIC COMPARISON ROUTINE THAT MAY BE USED IN
     CONNECTION WITH THE ABOVE. */

 DEFINEF ALPHBIGR (A, B);
 LOCAL N, CHARS, CHARPOS;
 /* DEFINE COLLATING SEQUENCE FOR CHARACTERS */
 CHARS =≠ ,ABCDEFGHIJKLMNOPQRSTUVWXYZ≠;
 CHARPOS = ≤ <CHARS(N), N>, 1 <= N <= #CHARS ≥;
 /* COMPARE EITHER FIRST DISTINCT CHARACTERS, OR LENGTHS */
 IF ∃ 1 <= N <= (#A) MIN, (#B) | A(N) NE, B(N)
     THEN RETURN CHARPOS(A(N)) GT, CHARPOS(B(N));
     ELSE RETURN (#A) GT, (#B); END IF;
 END ALPHBIGR;
 COMPLTE;
```

MISCELLANEOUS PERMUTATION ALGORITHMS

```
DO;
**G = COMPUTE; DO**

/* MAKE SEQUENCE OF TUPLE, */
DEFINEF MAKSEQ, TUP; RETURN{+:1<=N<=+TUP}≤:<N,TUP(N)>≥; END MAKSEQ,;

/* COMPUTES #HEAD# AS FUNCTION, */
DEFINEF HDD(X); RETURN HD,X; END HDD;

/* COMPOSES FUNCTIONS, */
DEFINEF   F C, G;  RETURN ≤ <X,G(F(X))> , X→ HDD [F] ≥ ;END F C,;

/* INVERTS A FUNCTION, */
DEFINEF   INV, F;  RETURN ≤ < HD,TL,X,HD,X>, X→F      ≥; END INV,

/* CYCLE FORM OF A PERMUTATION, */
DEFINEF   CYCFORM(F);
LOCAL  S,CYCS, CYC,ELT,E ;
/* FORM SET OF ALL ELEMENTS PERMUTED,  CHOOSE ONE OF THESE,
    AND REPEATEDLY APPLY PERMUTATION, UNTIL FULL CYCLE IS
    GENERATED,  ELEMENTS OF CYCLE ARE REMOVED FROM SET, AND THE
    PROCESS CONTINUES UNTIL NO ELEMENTS ARE LEFT IN THE SET,*/
S=HDD[F]; CYCS=NL,;
(WHILE S NE, NL, )
    ELT FROM,S ;   CYC = <ELT> ;
    (WHILE(F(ELT)IS,E ) → S  DOING  ELT = F(ELT);)
        CYC( +CYC +1)= E;   E OUT, S; END WHILE;
    CYC IN, CYCS;
    END WHILE  S;
RETURN CYCS;
END CYCFORM;
G;

/* INVERSE OF A PERMUTATION IN CYCLE FORM, */
DEFINEF  INVC, CYCS;
/* GIVEN A PERMUTATION IN CYCLE FORM, ITS INVERSE IS
    OBTAINED BY REVERSING EACH CYCLE, */
RETURN ≤[+:1<= N<= +C]<C((+C-N) + 1)>, C→CYCS≥;
END  INVC,    ;
G;
```

```
/* INVERSE OF A PERMUTATION. */

DEFINEF  CYCINV (F); LOCAL S,ELT,NEXT ;
/* THIS ALGORITHM RESEMBLES THE PRECEEDING ≠CYCFORM≠ PROCEDURE.
    HOWEVER, CYCLES ARE NOT FORMED EXPLICITLY, BUT RATHER ARE
    USED IMPLICITLY,  THE INVERSE PERMUTATION BEING BUILT
    UP BY MAPPING EACH ELEMENT INTO ITS PREDECESSOR IN THE CYCLE
    TO WHICH IT BELONGS. */
S= HD(F);
(WHILE S NE,NL,)
    ELT FROM, S; NEXT =F(ELT);
    (WHILE NEXT → S)
        NEXT OUT,S;
        <ELT,NEXT,FN,JK>  =<NEXT,F(NEXT),ELT>;
        F(ELT )=FN;
   END WHILE NEXT ;
   F(NEXT)=ELT; /*CLOSING THE LOOP*/
END WHILE S;
RETURN F;
END CYCINV;
COMPLTE;
```

PERMUTATION GENERATOR (O,P, VOL II, PAGE 142)

```
/* ON SUCCESSIVE CALLS, THIS ROUTINE GENERATES SUCCESSIVE
   PERMUTATIONS OF N ELEMENTS, IT RETURNS A PAIR OF THE FORM
   <PERMUTATION, FLAG>, WHERE FLAG IS ≠TRUE≠ UNLESS NO MORE
   PERMUTATIONS CAN BE GENERATED,
   IN WHICH CASE PERMUTATION=OM, */
/* IF CALED WITH MORE = F,, IT RE-INITIALISES USING N, AND
   RETURNS THE FIRST PERMUTATION ON N ELEMENTS, */

DO;
DEFINE PERM(N, MORE);
LOCAL K, J, KK, FIND, LM, TJ, JK;

     /* INITIALIZE IF NEW(MORE=F,) */
IF N, MORE THEN MORE=T,;TUPL=NULT, ;
     (∀1<=J<=N)TUPL(J)=J;;
     RETURN<TUPL,MORE>;
     END IF;

     /* IF TUPLE IS MONOTONE DECREASING THERE ARE
          NO MORE PERMUTATIONS, OTHERWISE FIND
          LAST POINT OF INCREASE */
IF N, (∃N> J >=1↑TUPL(J) LT, TUPL(J+1)) THEN
     MORE=F,;RETURN<OM,,MORE>;
     END IF;

     /* NEXT FIND THE LAST TUPL(K) WHICH EXCEEDS

         TUPL(J) AND SWAP */
FIND=∃N>= K >J↑TUPL(J) LT, TUPL(K);
<TJ, JK, LM> = <TUPL(K), TUPL(J)>;
TUPL(K)=JK;TUPL(J)=TJ;

     /* REARRANGE ALL THE ELEMENTS AFTER TUPL(J+1)
          INTO INCREASING ORDER */
(∀J<K<=((N+J+1)/2))
     KK=(N-K)+J+1;
     <TJ, JK, LM> = <TUPL(KK), TUPL(K)>;
     TUPL(KK)=JK;TUPL(K)=TJ;
END ∀ J;
RETURN<TUPL,MORE>;
END PERM;
COMPUTE;
```

HUFFMAN CODE ALGORITHMS (O,P, VOL II, PAGE 148)

```
/* THE FOLLOWING ROUTINES TRANSFORM A TABLE OF CHARACTER
    FREQUENCIES INTO A HUFFMAN ≠OPTIMAL≠ CODE TABLE, */

/* THE STRATEGY USED IS AS FOLLOWS,  FIRST A BINARY TREE,
    TO WHOSE TWIGS ALL THE CHARACTERS ARE ATTACHED, IS BUILT,
    THE CODE OF EACH CHARACTER IS THEN THE ADDRESS OF ITS TWIG
    RELATIVE TO TREE ROOT,  FOR EXAMPLE, A CHARACTER REACHED BY
    WALKING L-L-R-L-R-R-L FROM THE ROOT HAS THE CODE 0010110 */

/* DECODING (SEE THE ROUTINE ≠CSEQ≠ BELOW), IS ACCOMPLISHED BY
    STARTING AT THE TREE TOP AND USING SUCCESSIVE BITS OF A STRIN
    TO BE RECODED TO GOVERN LEFT AND RIGHT STEPS DOWN THE TREE
    UNTIL A TWIG IS REACHED,  THE CHARACTER AT THIS TWIG IS THE
    SYMBOL DECODED */

/* THE BINARY TREE IS BUILT AS FOLLOWS:
    THE TWO MINIMUM FREQUENCY CHARACTERS ARE FOUND AND MADE INTO
    THE IMMEDIATE DESCENDANTS OF A ≠COMPOSITE CHARACTER≠,
    OF FREQUENCY EQUAL TO THE SUM OF THE TWO FREQUENCIES, WHICH
    REPLACES THEM,  THIS CONTINUES UNTIL ONLY ONE CHARACTER
    REMAINS,  THIS CHARACTER IS THE ROOT NODE OF THE TREE
    BUILT UP BY THE ITERATIVE PROCESS THAT HAS BEEN
    DESCRIBED,  */


    /* AUXILIARY ROUTINE TO CHOOSE MINIMUM, */
DO;
DEFINEF GETMIN, SET;
LOCAL KEEP,LEAST,X;
/* TAKE AN ARBITRARY ELEMENT OF SET AND GUESS IT TO
    HAVE MINIMUM FREQUENCY, */
KEEP=ARB,SET ;
LEAST=WFREQ(KEEP);
/* NOW REPLACE THE ≠MINIMUM-TO-DATE≠ WITH ANY ELEMENT
    HAVING LOWER FREQUENCY, */
(∀X∈SET)
    IF WFREQ(X) LT, LEAST THEN
        KEEP=X;
        LEAST=WFREQ(X);
        END IF;
END ∀ X;
/* REMOVE THE MINIMUM FREQUENCY ELEMENT FROM THE WORKPILE
    USED BY THE MAIN ≠HUFTABLE≠ ROUTINE WHICH FOLLOWS, */
KEEP OUT, WORK;
RETURN KEEP;
END GETMIN,;
COMPUTE;
```

```
    /* ROUTINE TO PRODUCE HUFFMAN TREE AND CODE TABLE. */
    DO;
DEFINEF HUFTABL  (CHARS,FREQ);
LOCAL WORK,SEQ,WFREQ,L,R,C1,C2,N,CODE,SEQ,TOP;
/* START WITH COLLECTION OF ALL CHARACTERS, KNOWN FREQUENCIES,
    AND TREE WITH NO NODES. */
WORK=CHARS;WFREQ=FREQ;L=NL.;R=NL.;
(WHILE (#WORK) GT. 1)
/* FORM NEW NODE WHOSE DESCENDANTS ARE THE TWO EXISTING
    CHARACTERS OF MINIMUM FREQUENCY, WHICH BECOME ITS
    DESCENDANTS. */
    C1=GETMIN. WORK;C2=GETMIN. WORK;
    N=NEWAT.;L(N)=C1;R(N)=C2;
    WFREQ(N)=WFREQ(C1)+WFREQ(C2);
    /* NEW NODE IS ADDED TO LIST OF CHARACTERS (THE TWO OLD
        CHARACTERS HAVE BEEN REMOVED BY #GETMIN#). */
    N IN. WORK;
    END WHILE;
CODE=NL.;SEQ=NULT.;
/* CHARACTER CODES ARE FORMED BY ROUTINE #WALK#,
    WHICH DETERMINES ADDRESS OF EACH TWIG. */
TOP=ARB.WORK ;  WALK(TOP);
RETURN <CODE,L,R,TOP>;
END HUFTABL  ;
COMPLTE;


DO;
DEFINE WALK(TOP); /* RECURSIVE TREE-WALKER
WHICH BUILDS UP ADDRESS OF EACH TWIG*/
/* HUFTABLES EXTERNAL CODE,SEQ,L,R*/
/* TO BUILD UP ADDRESS, ADD A #0# FOR EACH STEP TO THE LEFT,
    AND A #1# FOR EACH STEP TO THE RIGHT. */
IF L(TOP) NE. OM. THEN
    SEQ=SEQ+<0>; WALK(L(TOP));
    SEQ=SEQ+<1>; WALK(R(TOP));
ELSE /*AT TWIG*/ CODE(TOP)=SEQ;
END IF;
/* BEFORE RECURSIVE RETURN, DELETE FINAL BIT OF NODE ADDRESS,*/
IF (#SEQ) GT. 0 THEN
    SEQ=SEQ(1:#SEQ-1);;
RETURN;
END WALK;
COMPLTE;


/* HUFFMAN DECODE ROUTINE. */
```

```
DO;
DEFINEF CSEQ(HUFTABS,SEQ);
/* SEE PRECEEDING COMMENT FOR EXPLANATION OF DECODE PROCESS. */
<JK,L,R,TOP,JK>=HUFTABS;
OUTPU =NULC,; NODE=TOP; N=1;
(WHILE N LE,#SEQ DOING N=N+1;)
    IF L(NODE)EQ. OM.  /*SO THAT WE ARE AT TWIG*/
        THEN OUTPU =OUTPU  + NODE ;
        NODE=TOP;N=N-1;
    ELSE IF SEQ(N) EQ. 0 THEN
        NODE=L(NODE);
    ELSE NODE=R(NODE);
    END IF;
END WHILE N;
RETURN OUTPU +NODE;
END CSEQ;
COMPUTE;
```

LINEAR TIME MEDIAN FINDING ALGORITHM

```
/*                              KTHONE                                  */

CO;
DEFINEF KTHONE(KPARAM, SETPARAM);

/* THE VALUE OF THIS FUNCTION IS THE KPARAM≠TH NUMBER, IN
ASCENDING ORDER, OF THE GIVEN SET ≠SETPARAM≠ OF NUMBERS.  IF
KPARAM IS OUT OF RANGE, THE RESULT IS UNDEFINED. */

/* THIS IS THE ALGORITHM DISCOVERED BY FLOYD, ET AL, IN LATE
1971, IT RUNS IN LINEAR TIME. */

LOCAL BIGPILE, CASE, I, K, MEDIAN, MIDPTS, SET, SMALPILE,
  U, V, X;
/* ≠KTHONEBL≠ IS A GLOBAL VARIABLE (TO PREVENT STACKING ON
RECURSION).  USER MUST INITIALIZE IT TO NULL CHAR. STRING. */

IF SETPARAM EQ. NL. THEN RETURN OM.;;

K = KPARAM;                 /* SAVE PARAMETERS (THIS ROUTINE */
SET = SETPARAM;             /* DOES NOT ALTER THEM).         */
KTHONEBL = KTHONEBL + ≠   ≠;  /* TO INDENT WHEN PRINTING
                                 NUMBER OF COMPARISONS. */
(WHILE (#SET) GE. 3)
  /* BUILD SET ≠MIDPTS≠, THE SET OF MIDDLE VALUES FROM
  ≠SET≠, TAKING THE NUMBERS THREE AT A TIME. */
  I = 2;
  MIDPTS = NL.;
  (ᵛX∈SET)
    I = (I+1)//3;
    IF I EQ. 0 THEN U = X;;
    IF I EQ. 1 THEN V = X;;
    IF I EQ. 2 THEN
      /* PUT MEDIAN OF U, V, AND THE CURRENT X INTO SET
      MIDPTS.  REQUIRES 3 COMPARISONS (WORST CASE). */
      IF X LT. V THEN CASE = 1; ELSE CASE = 0;;
      IF U LT. X THEN CASE = CASE + 2;;
      IF V LT. U THEN CASE = 3 - CASE;;
      /* NOW CASE MUST BE 1, 2, OR 3. */
      MIDPTS = MIDPTS WITH, (<U, V, X>)(CASE);;
      END IF I EQ. 2;
    END ᵛX;

  PRINT, KTHONEBL, (#SET/3)*3;  /* PRINT NUMBER OF COMPAR-
                                   ISONS, INDENTED. */
```

```
    /* AS MANY AS TWO MEMBERS OF ≠SET≠ HAVE NOT BEEN CONSIDERED
    FOR PLACEMENT IN ≠MIDPTS≠,  BUT THE ERROR IS NOT SUFFICIENT
    TO PREVENT THIS ALGORITHM FROM WORKING IN LINEAR TIME,
    NOTE THAT ↓MIDPTS GE, 1, BECAUSE ↓SET GE, 3, */
    /* NOW FIND THE (EXACT) MEDIAN OF ≠MIDPTS≠, IN LINEAR TIME,
    THIS ALGORITHM CHOOSES ON THE LOW SIDE IF ↓MIDPTS IS EVEN,*

    MEDIAN = KTHONE((↓MIDPTS+1)/2, MIDPTS);

    /* NOTE THAT ≠MEDIAN≠ IS SOMEWHERE IN THE MIDDLE THIRD OF
    ≠SET≠, */  /* PRECISELY, THE NUMBER OF MEMBERS OF ≠SET≠ THA
    ARE LESS THAN ≠MEDIAN≠ IS AT LEAST (N/3+1)/2 + (N/3+1)/2,
    AND THE NUMBER OF MEMBERS THAT ARE GREATER IS AT LEAST
    N/6 + (N/3+2)/2, WHERE N = ↓SET, */

    /* NOW DIVIDE ≠SET≠ INTO TWO PILES; MEMBERS OF ≠SMALPILE≠
    ARE LE, MEDIAN, AND MEMBERS OF ≠BIGPILE≠ ARE GT, MEDIAN, */

    SMALPILE = NL,;  BIGPILE = NL,;  /* INITIALIZE, */
    (∀X↓SET)
      IF X LE, MEDIAN THEN SMALPILE = SMALPILE WITH, X;
                      ELSE BIGPILE = BIGPILE WITH, X;;
      END ∀X;

    PRINT, KTHONEBL, ↓SET;  /* PRINT NUMBER OF COMPARISONS, */
    /* SINCE ↓SET GE, 3, AND WE HAVE THROWN THE MEDIAN INTO
    ≠SMALPILE≠, WE HAVE ↓SMALPILE GE, 2 AND ↓BIGPILE GE, 1,  NO
    ITERATE TO FIND THE APPROPRIATE MEMBER OF THE APPROPRIATE
    PILE, */

    IF K LE, ↓SMALPILE THEN SET = SMALPILE;
    ELSE SET = BIGPILE;  K = K - ↓SMALPILE;;
    END WHILE;  /* GO BACK WITH NEW SET AND POSSIBLY NEW K, */

KTHONEBL = KTHONEBL(1:↓KTHONEBL-3);
/* NOW ↓SET IS 1 OR 2 (IT CAN≠T BE ZERO),  K MAY BE OUT OF
RANGE IF THE ORIGINAL CALL HAD KPARAM OUT OF RANGE, */

IF (↓SET) EQ, 1 THEN
  IF K EQ, 1 THEN RETURN ARB, SET;
             ELSE RETURN OM,; END IF K;
ELSE /* ↓SET MUST BE 2, */
  IF K EQ, 1 THEN RETURN [MIN,: X↓SET] X;
  ELSE IF K EQ, 2 THEN RETURN [MAX,: X↓SET] X;
                  ELSE RETURN OM,; END IF;
END KTHONE;
COMPUTE;
```

TIME CHECK ROUTINE (PRINTS CURRENT CP TIME)

```
/*                          TIMECHEK                                    */

DO;
DEFINE TIMECHEK;
/* WRITES A TIME CHECK MESSAGE ON FILE ≠OUTPUT≠,  USER SHOULD
INITIALIZE THE GLOBAL VARIABLE ≠TIMEPREV≠ TO ZERO. */

LOCAL TIMENOW;
TIMENOW = TIME(0);  /* INVOKE BALM TIME ROUTINE. */
PRINT, ≠CP TIME (10THS SECS) = ≠, TIMENOW,
       ≠; TIME SINCE LAST CHECK = ≠, TIMENOW-TIMEPREV;
TIMEPREV = TIMENOW;
RETURN;
END TIMECHEK;
COMPUTE;
```

LEXICAL SCAN SETUP ROUTINE (O.P. VOL II, PAGE 108)

```
/* THE PROGRAM WHICH FOLLOWS REPRESENTS A LEXICAL SCAN
   METACOMPILER, WHICH ACCEPTS INPUT DATA DESCRIBING A FINITE-
   STATE AUTOMATON, AND A COLLECTION OF ≠SPECIAL ACTIONS≠
   TO BE PERFORMED IN PARTICULAR LEXICAL SITUATIONS;
   THE INPUT IS SYSTEMATICALLY CHECKED FOR CONSISTENCY, AND
   TRANSFORMED INTO A STATE TRANSITION TABLE ≠TABLE≠,
   A CHARACTER-TYPE FUNCTION ≠TYPEF≠,
   AND AN AUXILIARY ROUTINE PACKAGE ≠PAKTEXT≠; */
   /* THIS PROGRAM READS DATA FROM FILE ≠INFILE≠.  A SAMPLE SET
   OF INPUT DATA IS SHOWN BELOW, (DISCOUNT THE COMMENTING
   BRACKETS), */
/* STUDY OF THE FORM OF THIS DATA WILL HELP TO UNDERSTAND THE
   LOGIC OF THE PROGRAM WHICH FOLLOWS. */

   /* ≠ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789 ≠                    *
   /* <≠A≠ ≠0≠ ≠BL≠>                                             *
   /* ≤<≠A≠ <≠ABCDEFGHIJKLMNOPQRSTUVWXYZ≠>>                      *
   /*  <≠0≠ <≠0123456789≠>> <≠BL≠ <≠ ≠>>≥                        *
   /* ≤<≠NXT≠ <<≠GO≠ ≠NAME≠> <≠GO≠ ≠NUM≠> ≠SKIP≠>>              *
   /* <≠NAME≠ <≠CONT≠ <≠DO≠ ≠ZEROO≠ ≠CONT≠> ≠ENDO≠>>            *
   /* <≠NUM≠ <≠END≠ ≠CONT≠ ≠END≠>>≥                             *
   /* ≤<≠ZEROO≠ ≠NN=NN+1;ACTION=IF CSTRING(NN) NE. 0 THEN       *
   /*  ᵛᵛCONTᵛᵛ ELSE ᵛᵛENDᵛᵛ;≠>≥ */  /* ᵛᵛ = DOUBLE QUOTES,     *

   DO;
    **TYPE=TYPI**
   **SEQTYPE=SEQTYPES**
   INFILE=MAKFILE(≠INFILE≠, 72); /* ESTABLISH INPUT FILE */
   DEFINEF READCK(DUMMY);
   /* INPUT ROUTINE WHICH ≠ECHOES≠ INPUT */
    LOCAL X;
    READ,(X);
   PRINT,X;
               ELSE RETURN X;;
   END READCK;
   COMPUTE;


   **READCK=READCK(0)**
   /* MACRO MAKING IT UNECESSARY TO WRITE DUMMY ARGUMENTS.*/

   DO;
   DEFINEF SETUP(DUMMY);
   LOCAL NERRORS,SEQTYPES,CTYPES,TUP,TYPE,CLIST,N,
        CSTRING,J,C,TY,TU ,X,C,ALLC,TYPEF,RAWTABL,
        ROUTS,RT,ROUTSCLD,PAKTEXT,TXY,RNUMS,R,TXZ,
        STATSUSD,ST,Y,TABLE,STATE,TRIP,S2,S3,ROUTSET;
```

```
/* INITIALISE CHARACTER-TYPE FUNCTION TO BE EVERYWHERE
    UNDEFINED, */
TYPEF=NL,;
/* SET UP COLLECTION OF ALL CHARACTERS */
/* ≠ER≠ IS SPECIAL ≠END RECORD≠ CHARACTER */
ACSTR = READCK; ER=≠ER≠;
ALLC = {↑;1<=N< ↓ ACSTR} ≤ ; ACSTR(N) ≥WITH,ER;
/* THIS MACRO SETS UP THE TYPE OF A CHARACTER, ALLOWING
    MULTIPLE TYPES, WHICH ARE DIAGNOSED LATER, */
**SETYPE(C)=TYPEF ≤C≥ = TYPEF ≤C≥ WITH, TYPE**
/* INITIALISE THE NUMBER OF ERRORS TO ZERO;
    READ A TUPLE LISTING TOTAL FAMILY OF CHARACTER TYPES,
    AND ALSO READ COLLECTION OF PAIRS DECLARING TYPES OF
    PARTICULAR CHARACTERS, */
NERRORS=0;
SEQTYPES=READCK;
CTYPES=READCK;
(∀TUP↔CTYPES)
    <TYPE,CLIST,JK>=TUP;
    /* NOTE THAT ≠CYPTES≠ IS A SET OF PAIRS OF THE FORM
        <CHARACTER-TYPE,<TUPLE OF STRINGS CONTAINING
         CHARACTERS OF THIS TYPE> > */
    (∀1<=N<=↓CLIST)CSTRING=CLIST(N);
        IF CSTRING EQ,≠ER≠ THEN
           SETYPE (ER);
/* ≠ER≠ IS USED IN A SPECIAL WAY, AS EXTERNAL REPRESENTATION
OF AN INTERNAL ≠END RECORD≠ CHARACTER, */
        ELSE (∀1<=J<=↓CSTRING) C=CSTRING(J);
        SETYPE(C);;
        END IF CSTRING;
    END ∀1;
END ∀TUP;
**ERROR=NERRORS=NERRORS+1**
/* CHECK THAT RANGE OF TYPEF IS INDENTICAL WITH
    RANGE OF TYPESEQ */
TY = {↑;1<=N<=↓SEQTYPES}≤;SEQTYPES(N)≥;
 TU={↑;N↔TYPEF}≤; HD, TL, N≥;
X=TY-TU;IF X NE,NL, THEN
     PRINT,  ≠TYPES SPECIFIED BUT NOT USED ARE;≠,X;
     ERROR;;
X=TU-TY; IF X NE,NL,THEN
     PRINT,  ≠UNSPECIFIED TYPES ARE USED, THESE ARE;≠,X;
     ERROR;;
/* CHECK THAT NO UNANTICIPATED CHARACTERS APPEAR */
X = ≤ PR ↔ TYPEF ↑ N,(HD,PR) ↔ ALLC≥;
IF X NE, NL, THEN
   PRINT,  ≠UNANTICIPATED CHARACTERS APPEAR,THESE ARE;≠,X;
   ERROR;;
/*CHECK THAT ALL CHARACTERS HAVE UNIQUE TYPE SPECIFIED*/
```

```
X=SC→ALLC↑(↓TYPEF{C}) EQ,0};
IF X NE,NL, THEN
    PRINT, ≠TYPE UNSPECIFIED FOR FOLLOWING CHARACTERS:≠,X;
     ERROR;;
X=S C→ALLC↑(↓TYPEF{C})GT,1};
IF X NE,NL, THEN
PRINT, ≠TYPE MULTIPLY SPECIFIED FOR FOLLOWING CHARACTERS:≠,X;
    ERROR;;
/* READ IN RAW FORM OF LEXICAL STATE TRANSITION TABLE */
/* SEE SAMPLE DATA ABOVE FOR FORM OF DATA READ. */
RAWTABL=READCK;


DEFINEF HDD(X); RETURN HD,X; END HDD;

/* FORM COLLECTION OF ALL LEXICAL STATES MENTIONED IN DATA,
    THE STATE ≠NNT≠ DESIGNATING ≠NEXT TOKEN ABOUT TO BEGIN≠
    IS OBLIGATORY, */

STATSUSD=HDD {RAWTABL}; /*CHECK THAT ≠NXT≠ BELONGS
TO STATSUSD, AND THAT THERE ARE NO REPETITIONS*/
IF N, ≠NXT≠→STATSUSD THEN
     PRINT,  ≠REQUIRED STATE =NXT= OMITTED FROM TABLE≠;
    ERROR;;
X=SST→STATSUSD↑(↓RAWTABL{ST})GT,1};
IF X NE,NL,THEN
    PRINT, ≠MULTIPLY DEFINED STATES:≠,X;
    ERROR; /*FORCE TO SINGLE VALUED FUNCTION*/
    (∪Y→X) RAWTABL(Y)=ARB, RAWTABL{Y};;
END IF X NE, NL,;
/*CHECK THAT RIGHT NUMBER OF TERMS IN ALL
    ROWS OF TRANSITION TABLE, */
X=SST→STATSUSD↑(↓RAWTABL(ST)) NE, ↓SEQTYPES };
IF X NE, NL, THEN
PRINT, ≠STATES DEFINED WITH WRONG NUMBER OF TYPE ENTRIES:≠,X;
    ERROR;;
/*CONVERT TO MAP OF TWO INDICES*/
TABLE=S<STATE,SEQTYPE(J),RAWTABL (STATE)(J)>,
STATE→STATSUSD, 1<=J<=↓SEQTYPE};
/* COLLECTION OF KEYWORDS REQUIRING PARAMETERS, */
S2=S1≠GO≠,  ≠DO≠};
**TXY=TRIP(3)**
S3 =S;≠END≠,≠SKIP≠,≠CONT≠};
/* KEYWORDS NOT REQUIRING PARAMETERS, */
/* CHECK THAT PARAMETERS ARE PRESENT JUST WHERE THEY ARE
    CALLED FOR, */
X = STRIP*TABLE↑ ((N,PAIR, TXY) A, N, TXY→S3) OR
     PAIR, TXY A,N,(HD,TXY)→S2};
IF X NE,NL, THEN
PRINT, ≠ILLEGAL ENTRIES IN FOLLOWING POSITIONS OF TABLE≠,X;
```

```
    ERROR;;
/* CHECK THAT ALL PARAMETERS IN ≠GO≠ ENTRIES
    ARE VALID LEXICAL STATES. */
X=≤TRIP→TABLE↑(PAIR,TXY) AND,
     (HD,TXY) EQ,≠GO≠ AND, N, TXY(2)→ STATSUSD≥;
IF X NE,NL, THEN
    PRINT,
     ≠ILLFORMED GO TO ENTRIES IN FOLLOWING TABLE POSITIONS:≠
   , X;
  ERROR;;
/*NOW PREPARE TO CHECK WELLFORMEDNESS OF ALL CALL-TYPE
ENTRIES*/
/* READ IN COLLECTION OF LABELED, USER-DEFINED ROUTINES,
    FORM SET OF ALL LABELS USED, */
ROUTSET=READCK; ROUTS=HDD[ROUTSET];
/*CHECK THAT ALL ROUTINES UNIQUELY DEFINED*/
/* INITIALISE FOR SUBSEQUENT COLLECTION OF ALL ROUTINES
    MENTIONED IN TRANSITION TABLE, */
ROUTSCLD=NL.;
X=≤RT→ROUTS↑(↓ROUTSET≤RT≥)NE,1≥;
IF X NE,NL, THEN
    PRINT, ≠ILLDEFINED OR MULTIPLY DEFINED ROUTINES:≠,X;
    ERROR ;;
/* USING AUXILIARY ROUTINE ≠CALLOK≠, GIVEN BELOW,
    CHECK ON WELL-FORMEDNESS OF ALL ≠DO≠ TYPE TABLE ENTRIES *
X=≤TRIP→TABLE↑(PAIR,TXY) AND,
     (HD, TXY) EQ,   ≠DO≠ AND, N, CALLOK, TXY≥;
IF X NE,NL, THEN
PRINT, ≠ILLEGAL CALL-TYPE ENTRIES IN FOLLOWING POSITIONS:≠,X;
ERROR;;
/*CHECK THAT ALL ROUTINES CALLED ARE DEFINED*/
X=ROUTSCLD-ROUTS; IF X NE, NL, THEN
    PRINT,≠ROUTINES USED BUT NOT DEFINED ARE:≠,X;
    ERROR;;
/* CHECK THAT ALL ROUTINES DEFINED ARE ACTUALLY USED, */
X=ROUTS-ROUTSCLD;IF X NE,NL,THEN
PRINT,≠WARNING *-*-*- ROUTINES DEFINED BUT NOT USED:≠,X;;
/* NUMBER ROUTINES */
RNUMS = NL.; (∀R → ROUTS) RNUMS(R) = ↓ RNUMS +1 ;;
/* AT THIS POINT WE BEGIN TO PRODUCE A BLOCK OF VALID SETL
    CODE, ULTIMATELY TO CONSTITUTE THE AUXILIARY ROUTINE
    PACKAGE ≠RPAK≠. THIS CODE CONSISTS OF USER SUPPLIED CODE
    FRAGMENTS, MERGED WITH STANDARD ≠BOILERPLATE≠ */

/* THE CODE REQUIRED HAS THE FOLLOWING FORM
    DEFINEF RPAK(NUMROUT);
    -A COMMENT ON EXTERNAL VARIABLES USED-
    GO TO (ROUT1, ROUT2,. . . ,,ROUTN, ZZZZZZ> (NUMROUT);
    ROUT1; USER-SUPPLIED TEXT ,,,
    ROUT2; USER-SUPPLIED TEXT ,,
```

```
        ETC.
    RETURN;
    END RPAK;  */

/*SET UP RPAK FOR COMPILATION*/
RFINAL=≠ZZZZZZ≠;
/* DUMMY LABEL USED TO COMPLETE TUPLE */
PAKTEXT=≠DEFINE RPAK(NUMROUT);≠+
≠/*SETUP EXTERNAL CSTRING,TOKBEGIN, CURPOINTER, STATE,TOKEN,
DATA*/≠
+ ≠GO TO (<≠ +[+;ROUT → ROUTS] (ROUT + ≠,≠)
  + RFINAL + ≠>)(NUMROUT);≠ +
  [+;ROUT → ROUTS] (ROUT + ≠:≠ + ROUTSET(ROUT)
     + ≠RETURN;≠) + ≠END RPAK;≠;
/* NOW REPLACE ROUT NAMES IN ACTION TABLE
   BY CORRESPONDING INDEX IN RPAK */
(∀ X → STATSUSD, Y → TY + (PAIR, TABLE(X,Y))
    AND,(HD, TABLE(X,Y) EQ, ≠DO≠)
    TXZ=   TABLE(X,Y);
    J = 1;
(WHILE  (TXZ(J) IS, OP)NE,OM,
                 DOING J=J+1;)
        TXX=TXZ;
        IF OP EQ, ≠DO≠ THEN
                TXX(J+1) = RNUMS(TXX(J+1));
                TABLE(X,Y) = TXX;
        END IF;
   END WHILE;
END ∀X;
/* NOW RPAK HAS BEEN SET UP, TYPEF SUPPLIED,
     AND TABLE CONSTRUCTED */
RETURN<PAKTEXT,TYPEF,TABLE>;
END SETUP;
COMPUTE;


DO;
DEFINEF CALLOK, ENTRY;
/* AUXILIARY ROUTINE TO CHECK VALIDITY OF ≠DO≠ ENTRIES
    IN THE ACTION TABLE, */
/* SETUP EXTERNAL ROUTSCLD,STATSUSD; */
LOCAL OK,N,KEY,LABEL;
/* EXAMINE ALL ≠GO≠ AND ≠DO≠ ITEMS IN COMPOSITE TABLE, */
(∀ 1<=N<= + ENTRY + (ENTRY(N) IS, KEY) →
      ≤; ≠GO≠,  ≠DO≠≥)
/* ERROR IF FINAL ENTRY LACKS PARAMETER, */
    IF N EQ, + ENTRY THEN RETURN F.;;
```

```
/* ERROR IF ≠GO≠ TO NONEXISTENT LEXICAL STATE */
/* COLLECT NAMES OF ALL ROUTINES IN DO ENTRIES. */
IF KEY EQ. ≠GO≠ THEN IF N. ENTRY(N+1) → STATSUSD
     THEN RETURN F.;
     ELSE /*KEY=CALL */
     ENTRY(N+1) IN. ROUTSCLD;
END IF KEY;
END ↓1<=;
RETURN T.;
END CALLOK.;
COMPUTE;
```

```
    TREE PRINTING ROUTINES


/* THE PACKAGE OF ROUTINES WHICH FOLLOWS PRINTS A TREE
    IN A TWO-DIMENSIONAL TREE-LIKE FORMAT, WITH THE ROOT
    AT THE LEFT-HAND EDGE OF THE PAGE, AND PARENTS CONNECTED
    BY ARROWS TO THEIR OFFSPRING */
   DO;
/* THE ROUTINE WILL HANDLE BOTH BINARY AND ORDERED TREES */
/* LARGE TREES ARE RECURSIVELY BROKEN UP INTO SUBTREES
    WHICH ARE SMALL ENOUGH TO PRINT, */

/* WE BEGIN WITH VARIOUS AUXILIARY ROUTINES, */

/* THIS ROUTINE CONVERTS A NODE TO BE PRINTED INTO AN
    EXTERNAL FORM OF AT MOST 30 CHARACTERS LONG, */


  DEFINEF STRINGF(OBJ);
  LOCAL X;
  X=STRINGOF(OBJ);
  IF(#X)LE,30 THEN RETURN X;
  ELSE RETURN X(1:30);;
  END STRINGF;

    DEFINEF PADOUT(LINE);
    /* THIS PADS A LINE WITH BLANKS TO SPECIFIED LENGTH */
   RETURN LINE+MBLANKS(1;LINLIM-#LINE); END PADOUT;
   COMPUTE;




   DO;
   DEFINE PRNTIN(NODE);
   /*PRINTS NODE WITH ALL PREFIXED ARROWS, AND
      WITH ADDRESS IF IT IS TOP OF SUBTREE*/
   /*TPRINT EXTERNAL AUXSEQ,SEQNO*/
   LOCAL LINE,AUX,N;
    /* THE VECTOR AUXSEQ IS SET UP BY THE MASTER ROUTINE
      PRNTOUT, GIVEN BELOW.  IT DESCRIBES THE SET OF UP AND DOWN
      ARROWS, ETC., TO BE PREFIXED TO THE NODE APPEARING ON A
      GIVEN LINE.
     THE CODE +2 OR +3 DESIGNATES AN * IN GIVEN COLUMN.
      CODE -2 OR -3 INDICATES A ↓,
      CODE +1 INDICATES /→ , POINTING TO A RIGHT DESCENDANT,
      CODE -1 INDICATES ↓→, POINTING DIRECTLY TO LEFT DESCENDANT.
    */
   IF(#AUXSEQ) EQ,1 THEN GO TO ISTOP;;
   LINE= *       *;
```

```
      (∀1<=N<↓AUXSEQ-1)
          AUX=AUXSEQ(N);
          IF AUX GE 2 THEN LINE=LINE+≠ ↑ ≠;
          ELSE IF AUX LE -2 THEN LINE=LINE+ ≠ ↓ ≠;
          ELSE LINE=LINE+≠   ≠;
          END IF;
           END ∀1;
      AUX=AUXSEQ(↓AUXSEQ-1);
      IF AUX EQ 1 THEN LINE=LINE+ ≠ /→≠;
           ELSE LINE = LINE+ ≠ ↓→≠;
          END IF;
      /*NOW PRINT ITEM ITSELF*/
      PRINT,   PADOUT(LINE+STRINGF(S(NODE)));
      RETURN;
ISTOP:/*HERE THE TOP NODE, WITH ITS NUMBER, IS TO BE PRINTED*/
       /* THE TOP NODE A TREE OR SUBTREE IS PREFIXED WITH
          !N:, N BEING THE TREE SERIAL NUMBER (ALSO ESTABLISHED
          BY PRNTOUT).  THIS SERIAL NUMBER IS USED TO REFERENCE
          CROSS-REFERENCED SUBTREES, */
      LINE=DEC, SEQNO;LINE=≠;≠+((4-↓LINE)*≠ ≠)+LINE+≠;≠;
      PRINT, PADOUT (LINE+STRINGF(S(NODE))); RETURN;
      END PRNTIN;
      COMPUTE;


      DO;
      DEFINE PRNTNO(NUM);
      /*WORKS MUCH LIKE PRNTIN,BUT AUXSEQ 1 SHORTER,
           AND PRINTS   ;(CONVERTED NUMBER) RATHER THAN S */
       /* THIS ROUTINE USED WHEN A REFERENCE TO A SUBTREE
          RATHER THAN A NODE IS TO BE PRINTED, */
      /*TPRINT EXTERNAL AUXSEQ*/
      LOCAL LINE,AUX,N;
      LINE= ≠      ≠;
      (∀1<=N<↓AUXSEQ) AUX=AUXSEQ(N);
          IF AUX GE 2 THEN LINE=LINE+ ≠ ↑ ≠;
               ELSE IF AUX LE -2 THEN LINE=LINE+ ≠ ↓ ≠;
               ELSE LINE=LINE+≠   ≠;
          END IF;
      END ∀1;
      AUX=AUXSEQ(↓AUXSEQ);
      IF AUX EQ 1 THEN LINE=LINE+≠ /→≠;
           ELSE LINE=LINE+ ≠ ↓→≠;
           END IF;
      PRINT,  PADOUT(LINE+≠;≠+DEC, NUM); RETURN;
      END PRNTNO;
      COMPUTE;
  DO;
```

```
      DEFINEF AUXNEED(TOP);
      /*CALCULATES THE NUMBER OF COLUMNS TO THE RIGHT
           WHICH ARE NEEDED IF AN ITEM AND ITS IMMEDIATE
           DESCENDANTS ARE TO BE PRINTED IN PLACE*/
      /*TPRINT  EXTERNAL L,R,D,S,BIN*/
      LOCAL DESCS,N;
       /* FLAG ≠BIN≠ SET WHEN BINARY TREES, RATHER THAN ORDERED TREE
          ARE BEING PROCESSED. */
       /* THE PARENT/CHILD RELATION IS GIVEN BY TWO FUNCTIONS ≠L≠
          AND ≠R≠ ( LEFT AND RIGHT DESCENDANTS) FOR BINARY
          TREES.  FOR ORDERED TREES,  DESCS(NODE)(N) IS THE
          N-TH DESCENDANT OF NODE ≠NODE≠. */
      IF BIN THEN GO TO BINCAS;;
       /* NOTE THAT THE FUNCTION ≠S≠ ASSOCIATES WITH EACH TREE-
          NODE, A REPRESENTATION TO BE PRINTED. */
      DESCS=D(TOP);
      IF DESCS EQ. OM. THEN RETURN +STRINGF(S(TOP));
          ELSE RETURN +STRINGF(S(TOP)) MAX.
           ([MAX,: 1<=N<=+DESCS](+STRINGF(S(DESCS(N)))+3));
      END IF;
       /* SINCE DESCENDANT NODE INDENTED THREE CHARACTERS*/
BINCAS:     IF L(TOP) EQ. OM. THEN
                IF R(TOP) EQ. OM. THEN RETURN +STRINGF (S(TOP));
               ELSE RETURN
                    +STRINGF(S(TOP)) MAX.(+STRINGF(S(R(TOP)))+3);
               END IF R;
            ELSE IF R(TOP) EQ.OM. THEN
            RETURN +STRINGF(S(TOP)) MAX.(+STRINGF(S(L(TOP)))+3);
            ELSE RETURN +STRINGF(S(TOP)) MAX.
              (+STRINGF(S(L(TOP)))+3) MAX.(+STRINGF(S(R(TOP)))+3);
          END IF L;
      END AUXNEED;
      COMPUTE;



      DEFINE TPRINT.  X;
      /* THIS IS THE MASTER TREE-PRINT ROUTINE,*/
      LOCAL L,R,S,TOP,D,AUXLEN,PRPILE,AUXSEQ,PRTUP,MINS,NTOP,
           SKIP,JK,GENNO;
       /* THE ASSUMED FORM OF THE ARGUMENT TO THIS ROUTINE
          IS AS FOLLOWS: */
      /*X=<L,R,S=FUNCTION ASSOCIATING THING TO PRINT WITH NODE, TOP>
                  IF BINARY TREE*/
      /*X=<DESC,S,TOP> IF ORDERED TREE */
      SKIP=≠ ≠;
       /* EXTRACT PARAMETERS FROM ARGUMENT TUPLE.
```

```
    SET OR DROP ≠BINARY≠ FLAG, */
IF(↓X)EQ,4  THEN <L,R,S,TOP,JK>=X; BIN=T,;
     ELSE <D,S,TOP,JK>=X; BIN=F,; END IF;
 GENNO=0;
 /* GENNO SERVES TO GENERATE SERIAL NUMBERS FOR SUBTREES,
   AS NEEDED. PRPILE IS PILE OF SUBTREES WAITING TO BE
    PRINTED. */
PRPILE =≤;<TOP,0> ≥; /*THE ITEMS ON PRPILE ARE
<TOP OF SUBTREE TO PRINT, SERIAL NUMBER OF SUBTREE>*/
(WHILE PRPILE NE,NL,)
 /* NOTE THAT PRNTOUT WILL GENERATE NEW SUBTREES IF THE
    TREE IT IS PROCESSING IS TOO BIG TO PRINT. */
     PRTUP FROM, PRPILE;
    <NTOP,SEQNO,JK>=PRTUP;
    PRINT, SKIP; PRINT, SKIP; /*THUS SKIPPING 2 LINES*/
    AUXSEQ=NULT,; PRNTOUT(NTOP);
 END WHILE;
 RETURN;
END TPRINT,;
COMPUTE;


DC;
DEFINE PRNTOUT(NTOP);
LOCAL N,DESCS,NTOLAST;
 /* THIS IS THE PRINCIPAL EXECUTIVE ROUTINE OF THE TREE-PRINT
     PACKAGE.  IT GENERATES SUCCESSIVE ≠LINES≠ OF THE TREE
     TO BE PRINTED, AND CALLS ≠PRNTIN≠ TO DO THE ACTUAL
     PRINTING.  ≠NTOP≠ IS THE TOP NODE OF A TREE TO BE
     PRINTED.  ≠AUXNEED≠ REPRESENTS THE ENCODED FORM OF
     A LINE. */
/*TPRINT EXTERNAL BIN,D,L,R,GENNO,AUXSEQ*/
IF(AUXNEED(NTOP)+ 3*↓AUXSEQ*3) GT, LINLIM THEN GO TO ISBIG;;
IF BIN THEN GO TO BINCAS;;
DESCS=D(NTOP); IF DESCS EQ, OM, THEN GO TO TWIG;;
 /* ENTRY IS MADE HERE WHEN A DESCENDANT NODE WHICH IS
     NOT A TREE-TWIG IS TO BE PRINTED.*/
 /* THE VECTOR ≠AUXSEQ≠ KEEPS TRACK OF THE TREE-POSITION
     OF A GIVEN NODE, AND IS USED BY ≠PRNTIN≠ TO SET UP A
     PATTERN OF ARROWS ON THE LEFT-HAND PORTION OF THE LINE ON
     WHICH A NODE APPEARS.*/
 /* AUXSEQ IS ESSENTIALLY THE TREE ADDRESS OF A NODE,
     REPRESENTED AS A TUPLE OF INTEGERS, A POSITIVE INTEGER
     REPRESENTING A LEFT-HAND DESCENDANT (POSITIONED HIGHER ON
     THE PRINTED PAGE) AND A NEGATIVE INTEGER REPRESENTING A
     RIGHT-HAND DESCENDANT (POSITIONED LOWER ON THE
     PRINTED PAGE). */
 /* NOTE THAT FOR A NODE WITH N DESCENDANTS IN AN ORDERED
     TREE, DESCENDANTS 1 THRU (N+1)/2 ARE REGARDED AS
     LEFT-HAND, AND THE REMAINING DESCENDANTS AS
```

```
        RIGHT-HAND, */
      /* HOWEVER, THE CODING USED IN AUXSEQ IS SOMEWHAT COMPLICATED
        BY THE NEED TO TRANSMIT ADDITIONAL INFORMATION TO ≠PRNTIN
        SO THAT AN APPROPRIATE PATTERN OF RISING AND FALLING
        ARROWS CAN BE FORMED.
        FIRST, CONSIDER A BINARY TREE,
        IF NODE1 HAS NODE2 AS A LEFT-HAND DESCENDANT, THEN
        ALL THE RIGHT-HAND DESCENDANTS OF NODE2 WILL ≠LIE UNDER
        THE BRANCH≠ CONNECTING NODE1 TO NODE2, AND THEREFORE CALL
        FOR A ↑ AT THE LEVEL OF NODE1.  THE SAME HOLDS WITH LEFT/
        RIGHT AND ↑/↓ REVERSED.  ACCORDINGLY, WE MARK EVERY
        LEFT-RIGHT OR RIGHT-LEFT REVERSAL IN AUXSEQ BY
        +3 OR -3 INSTEAD OF +1 OR -1 AS A COMPONENT, THUS
        FLAGGING FOR THE PRINTING OF ↑ OR ↓ */
      /* IN THE CASE OF AN ORDERED TREE THERE IS ANOTHER SLIGHT
        COMPLICATION TO BE FACED, THE OFFSPRING OF A ≠NON-EXTREME
        LEFT (OR RIGHT) DESCENDANT ≠LIES UNDER THE BRANCH≠
        LEADING TO MORE EXTREME LEFT (OR RIGHT) SIBLINGS OF ITS
        PARENT.  ACCORDINGLY, +2 AND -2 ARE USED TO SIGNAL
        NONEXTREME DESCENDANTS, AND TO CAUSE ↑ OR ↓ TO
        BE PRINTED, */
      /* PRINT LEFT DESCENDANTS FIRST, IF CURRENT NODE IS EXTREME
        RIGHT DESCENDANT, SET ITS PATH TO SHOW EVEN THOUGH IT
        IT WOULD NOT IF THERE WERE NO REVERSAL, */
      IF AUXSEQ EQ. NULT, THEN GO TO SKIP1;;
      IF AUXSEQ(↓AUXSEQ) EQ.-1 THEN AUXSEQ(↓AUXSEQ)= -3;;
      /*LEFTMOST DESCENDANT*/
SKIP1: AUXSEQ=AUXSEQ+<1>; /*1 DESIGNATES LEFTMOST*/
      PRNTOUT(DESCS(1));
      AUXSEQ(↓AUXSEQ)=2;
      /* 2 DESIGNATES NONEXTREME LEFT DESCENDANTS, */
      /* PRINT REMAINING LEFT-HAND DESCENDANTS*/
      (*1<N<=( ↓DESCS+1)/2)PRNTOUT(DESCS(N));;
      /* NOW PRINT NODE ITSELF, */
      PRNTIN(NTOP);
      AUXSEQ(↓AUXSEQ)=-2;
      /*-2 DESIGNATES NON-EXTREME RIGHT DESCENDANTS, */
      /* NOW BEGIN TO PRINT RIGHT DESCENDANTS, */
      NTOLAST=OM,;
      /* AUXSEQ WILL HAVE ONE COMPONENT IF AT TOP OF TREE, */
      IF(↓AUXSEQ) NE. 1
         THEN NTOLAST=AUXSEQ(↓AUXSEQ-1);;
      /* CORRECT -3 SETTING, IF IT PERSISTS, AND CHANGE 1 TO 3
           IN NEXT-TO LAST LAST COMPONENT OF NODE ADDRESS VECTOR, */
      IF NTOLAST EQ. -3
          THEN AUXSEQ(↓AUXSEQ-1)=-1;
          ELSE IF NTOLAST EQ.1
          THEN AUXSEQ(↓AUXSEQ-1)=3;
      END IF NTOLAST ;
      /* IN CASE OF A SINGLE DESCENDANT, WHICH IS TREATED AS LEFT,
```

```
       THERE ARE NO RIGHT DESCENDANTS, */
      IF(#DESCS) EQ.1 THEN GO TO DONE;;
      /* OTHERWISE PRINT ALL NONEXTREME RIGHT DESCENDANTS. */
      (W(#DESCS+1)/2<N<#DESCS) PRNTOUT(DESCS(N));;
      /* AND THEN PRINT RIGHTMOST DESCENDANT*/ AUXSEQ(#AUXSEQ)=-1;
      PRNTOUT(DESCS(#DESCS));
DONE; IF(#AUXSEQ) EQ. 1 THEN AUXSEQ=NULT,; RETURN; END IF;
      /* IF NECESSARY, CORRECT PRIOR LEFT TO RIGHT ≠TURN≠ FLAG,*/
      IF AUXSEQ(#AUXSEQ-1) EQ. 3 THEN AUXSEQ (#AUXSEQ-1)=1; ;
DONER; IF(#AUXSEQ)EQ. 1 THEN AUXSEQ=NULT,; RETURN; END IF;
      /* CUT OFF FINAL COMPONENT OF ADDRESS BEFORE RECURSIVE RETURN*
      AUXSEQ=AUXSEQ(1:#AUXSEQ-1); RETURN;
      /* ENTER HERE FOR TREATMENT OF BINARY TREES, */
BINCAS;AUXSEQ=AUXSEQ+<1>; IF L(NTOP) EQ. OM, THEN GO TO NOLEFT;;
      IF(#AUXSEQ)EQ. 1 THEN
          PRNTOUT(L(NTOP));
          GO TO NOLEFT;
      END IF;
      /* IF NOT TOP, AND PRIOR IS RIGHT DESCENDANT, SIGNAL ≠TURN≠
           BEFORE PRINTING LEFT-DESCENDANT SUBTREE, */
      IF AUXSEQ(#AUXSEQ-1) EQ.-1 THEN AUXSEQ(#AUXSEQ-1)=-3;;
      PRNTOUT(L(NTOP));
      /* THEN PRINT NODE, AND SIGNAL FOR RIGHT DESCENDANT. */
NOLEFT;PRNTIN(NTOP); AUXSEQ(#AUXSEQ)=-1;
      /* IF NECESSARY, CORRECT PRIOR RIGHT TO LEFT
           ≠TURN≠ FLAG, */
       NTOLAST = OM,  ;
          IF(#AUXSEQ)NE. 1
               THEN NTOLAST = AUXSEQ( #AUXSEQ -1);;
          IF NTOLAST EQ.-3  THEN AUXSEQ(#AUXSEQ-1)=-1;
          /* AND IF NECESSARY, SET LEFT TO RIGHT ≠TURN≠ FLAG,*/
          ELSE IF NTOLAST EQ.1 THEN AUXSEQ(#AUXSEQ-1)=3;;
          IF R(NTOP) EQ. OM, THEN GO TO DONE;;
      /* PRINT RIGHT DESCENDANT TREE IF ANY, */
      PRNTOUT(R(NTOP));GO TO DONE;
TWIG; /*NON-BINARY TWIG CASE*/
      AUXSEQ=AUXSEQ+<0>;
      /* A DUMMY FINAL COMPONENT, SINCE PRNTIN ASSUMES THAT
           #AUXSEQ EQ. N+1  WHEN AN N-TH LEVEL NODE IS PRINTED, */
      PRNTIN(NTOP);GO TO DONER;
      /* SEQUENCE FOR TREATING SUBTREE TOO EXTENDED TO FIT,
           GENERATE SEQUENCE NUMBER FOR SUBTREE; PRINT ITS INDEX
           INSTEAD OF ITS TOP NODE; PUT SUBTREE INTO WORKPILE
           FOR FUTURE PRINTING, */
ISBIG;GENNO=GENNO+1; PRNTNO(GENNO);
     (<NTOP,GENNO>)IN, PRPILE;
      RETURN;
      END PRNTOUT;
```

```
COMPUTE; FINISH;
```

NODAL SPAN PARSING ALGORITHM: (O.P. VOL II, P.158)

```
  /* NOTE: THE INPUT IS AT THE END OF THE PROGRAM*/
DO;
 DEFINEF NODPARS (INPU,GRAM,ROOT,SYNTYPES);

 /* THE ROUTINE WHICH FOLLOWS REPRESENTS JOHN COCKE≠S NODAL SPAN
     PARSING ALGORITHM, SEE THE CITED SECTION OF THE O.P. VOL II
     FOR A DETAILED ACCOUNT OF THIS ALGORITHM, AND VARIOUS
     IMPROVEMENTS OF IT DUE TO JAY EARLEY AND OTHERS. */
 /* THE ARGUMENTS TO THIS ROUTINE ARE AS FOLLOWS;
     INPUT - IS A TUPLE, REPRESENTING A TOKEN STRING TO BE PARSED.
     GRAM - IS A GRAMMAR IN ≠CHOMSKY NORMAL FORM≠, I.E.,
         CONTAINING BINARY PRODUCTIONS ONLY. EACH PRODUCTION
         A → BC   IS REPRESENTED BY A TRIPLE  <B, C, A > ,
     SYNTYPES - IS A MAP SENDING EACH TOKEN INTO THE SET OF ALL
         POSSIBLE SYNTACTIC TYPE SYMBOLS WHICH CAN REPRESENT IT,
     ROOT - IS THE ROOT SYMBOL OF THE GRAMMAR, */
 /* THE APPROACH IS AS FOLLOWS :  IF THE N-TH THRU
     (M-1)-ST SYMBOLS OF INPUT CAN BE PARSED AS AN ELEMENT OF
     SOME SYNTACTIC TYPE ≠A≠, THEN THE SPAN <M,A,N> IS SAID TO
     BE PRESENT IN THE INPUT, */
 /* THE SET OF ALL SPANS PRESENT CAN BE FOUND BY,A BINARY
     TRANSITIVE-CLOSURE LIKE PROCESS WHICH COMBINES
     ADJACENT SPANS, */
 /* INPUT IS GRAMMATICAL IF AND ONLY IF THERE IS A SINGLE
     SPAN COVERING THE WHOLE OF IT, AND IF THE SECOND COMPONENT
     OF THIS SPAN IS THE GRAMMAR≠S ROOT SYMBOL, */
 /* IF SUCH A SPAN EXISTS, CALL IT THE ≠TOPSPAN≠,
     A SPAN IS ≠RELEVANT≠ IF IT IS INVOLVED (ANCESTRALLY) IN THE
     CONSTRUCTION OF TOPSPAN,  THE ALGORITHM GIVEN BELOW FINISHES
     BY CONSTRUCTING RELEVANT SPANS, KEEPING THEM, AND
     DISCARDING IRRELEVANT SPANS, */

 LOCAL TODO,S,N,NEXT,EN ,TYP,M10,
     SPEND,TYP1,TYP2,NEWSP,TOPSPAN,D,X;
 /* ≠TODO≠ IS A WORKPILE OF SPANS WAITING TO BE COMBINED WITH
     THEIR LEFT-HAND NEIGHBORS, */
 /* DIVLIS RECORDS THE MANNER IN WHICH EACH SPAN WAS PRODUCED, */
 TODO=NL,;DIVLIS=NL,;
 /* INITIALISE ≠SPANS≠ TO INCLUDE EVERY ONE-LETTER SPAN
     COVERING THE FIRST INPUT TOKEN, */
 SPANS= ≤ <2,S,1>, S=SYNTYPES≤INPU(1)≥≥;
 (∀1<N≤#INPU)
 /* WORKING FROM LEFT TO RIGHT, GENERATE NEW ONE-SYMBOL SPANS,
     AND COMBINE THEM WITH ADJACENT SPANS TO THE LEFT, REPEATING
     AS LONG AS NEW COMBINATIONS ARE GENERATED, */
     TODO = ≤ <N+1,S,N> ,S= SYNTYPES≤INPU(N)≥ ≥;
     SPANS=SPANS+TODO;
```

```
      (WHILE TODO NE, NL,)
          NEXT FROM, TODO;

/* THE FOLLOWING MACRO CONSTRUCTS NEW SPANS AND PLACES THEM IN
SPANS AND IN TODO*/
 /* IT ALSO ADDS ITEMS TO THE DIVLIS ≠ANCESTRY RECORD≠
      AS APPROPRIATE, */

 **MAKNEW(NEXT)= <EN ,TYP2 ,MID>=NEXT; MID=HD, MID;
 /* COMBINE SPAN  <LAST, TYP2, MID > WITH < MID, TYP1, START>
      IF GRAMMAR ALLOWS, GENERATING NEW SPANS AND MAKING NEW
      ≠ANCESTRY RECORD≠ ENTRIES, */
 (* SPEND→ SPANS≤MID≥ , TYP → GRAM ≤(HD, SPEND) IS,TYP1,TYP2≥)
     NEWSP=<EN,TYP,HD,(TL,SPEND)>;
     DIVLIS=DIVLIS WITH,<NEWSP,MID,TYP1,TYP2>;
      IF N, NEWSP → SPANS THEN
          NEWSP IN, SPANS;
          NEWSP IN,TODO;
      END IF;
END *SPEND;
 **
      MAKNEW (NEXT)
      END WHILE TODO;
 END * 1<N;
 /* NOW CHECK GRAMMATICALITY OF INPUT STRING */
 IF N, (<↓INPU↑1,ROOT,1> IS,TOPSPAN) → SPANS THEN
     RETURN <NL,,NL,,F,> ;;
      /*ELSE CLEAN UP SET OF SPANS AND DETERMINE AMBIGUITY */
 PRINT, ≠SPANS BEFORE CLEANING≠, SPANS;
 PRINT, ≠DIVISION  LIST BEFORE CLEANING≠, DIVLIS;
 /* THROW AWAY SPANS,  RELEVANT ONES WILL BE RECOVERED BY
      ≠GETDESCS≠, */
 /* ≠AMB≠ IS A FLAG SET TO TRUE IF PARSE IS AMBIGUOUS, */
 SPANS=NL,;AMB=F,;GETDESCS(TOPSPAN);

/* THE FOLLOWING RETAINS IN THE ANCESTRY RECORD ONLY THOSE TUPLES
     WHOSE FIRST COMPONENTS ARE IN THE CLEANED SPANS  */

      DIVLIS = ≤ D→DIVLIS↑(HD, D)→ SPANS≥;
      RETURN< SPANS,DIVLIS,AMB>;
      END NODPARS;


/* THE FOLLOWING SUBROUTINE, BY TRACING SPAN ANCESTRY FROM THE ROO
SYMBOL DOWN, ELIMINATES UNNECCESSARY SPANS PRODUCED
      BY MACRO MAKNEW, */

      DEFINE GETDESCS(TOP);
      /*AUXILIARY ANCESTRY-TRACING SUBROUTINE*/
      LOCAL EN ,START,MID,TYP1,TYP2,X,JK;
```

```
      IF TOP↛SPANS THEN RETURN;;
      TOP IN, SPANS;
      /* PARSE IS AMBIGUOUS IF SOME RELEVANT SPAN HAS
          MULTIPLE ANCESTRY, */
      IF(↓DIVLIS≤TOP≥ )GT, 1 THEN AMB= T,;;
      /* EXTRACT COMPONENTS OF SPAN TO RECONSTRUCT ANCESTRY, */
      <EN ,JK,START,JK> = TOP;
          (∨X↛DIVLIS≤TOP≥)
           <MID,TYP1,TYP2,JK> = X;
           GETDESCS (<EN ,TYP2,MID>);
           GETDESCS(<MID,TYP1,START>));
  END ∨ X;
      RETURN;
      END GETDESCS;
      COMPUTE;
```

```
DC;

/*  THE FOLLOWING IS SAMPLE INPUT TO THE NODAL PARSE ROUTINE
A PRODUCTION A→BC IS WRITTEN AS A TUPLE <≠B≠,≠C≠,≠A≠>;
ALL PRODUCTIONS ARE INCLUDED IN THE SET GRAM, THEY MUST BE IN
CHOMSKY NORMAL FORM, THESE PRODUCTIONS CAN NOT INCLUDE TERMINALS O
RIGHT HAND SIDE ( NOR OF COURSE ON THE LEFT) TERMINALS  ARE
IN SYN*/

 GRAM={<≠A≠,≠B≠,≠S≠>,<≠A≠,≠B≠,≠A≠>};PRINT, ≠GRAMMER≠, GRAM;

/* CN IS THE INPUT STRING
    IN THIS PARTICULAR CASE  THE INPUT STRING IS XYZ,*/

CN = <≠X≠, ≠Y≠, ≠Z≠ >; PRINT, ≠INPUT STRING≠, CN;

/* ≠SYN≠ IS A MAP SENDING EACH INPUT TOKEN TO THE SET OF ALL
     ITS POSSIBLE SYNTACTICAL TYPES */

SYN={<≠X≠,≠A≠>,<≠Y≠,≠B≠> };PRINT,≠SYNTYPES≠, SYN;
```

09,49,08,LP14 3,273 KLN.

Item 12. DESCRIPTION OF THE SETL LANGUAGE.

First Part: Object Types, Expressions.

1. Introduction

A programming language may be judged by the data structures which it incorporates, which should ideally be useful in a wide variety of circumstances and permit at least moderately efficient manipulation. In presenting the SETL language which is now to be described we are suggesting that general finite sets can be represented and manipulated in useful ways.

Much of the expression semantics of SETL is modeled upon that used in the mathematical theory of sets, and many of the syntactic conventions used reflect notations which are standard in that theory. It is therefore worth making a few comments on mathematical set theory itself before immersing ourselves in the linguistic details of SETL. (These comments will be of more interest to those readers with a mathematical background than to those principally interested in programming languages.) An elegant short summary of the axiomatic foundations of the subject appear in Cohen [1], pp. 52-56. Axiomatic set theory uses a very small number of formal primitives, in terms of which the whole structure of mathematics can be built up rapidly and comfortably. We list these set theoretic primitives, and comment on the SETL constructions which correspond to them; of course, whereas mathematical set theory deals with both finite and infinite sets, SETL, intended as an executable programming language, deals with finite sets only. The primitives in question are:

(a) The *null set*. Provided as a basic entity in SETL.

(b) For two sets a and b, the *unordered pair* {a,b}.

This set-by-enumeration primitive is provided as a basic construction in SETL, even in a somewhat generalized form. It obviously yields a set (of two elements), thus conforming to the desire of of pure set theory to avoid the introduction of objects other than sets. We may note in this connection that SETL will from the start take a somewhat different path from pure set theory, since its

semantic universe will include not only sets but also atoms.

oms are customarily excluded from pure set theory on grounds of elegance. This exclusion, amounting to the insistence that set theory deal only with sets and not with objects of any other kind, is inconvenient in a programming language context, where one generally wishes for practical reasons to be able to speak of integers and character strings at least, without being forced to the trouble of mapping these additional objects upon objects of pure set theory. Thus for programming purposes a set theory including atoms which themselves have no members but which may freely be members of sets is more convenient than pure set theory. SETL incorporates this convenient modification. The atoms of SETL will generally be 'primitive' data types, such as integers, character strings, and so forth, familiar from conventional programming languages; we will find it convenient however to consider 'tuples' to be non-sets also, in a manner to be explained below.

Sets may have sets or atoms as members; two sets A and B are equal if and only if each member of A is equal to some member of B, and conversely. Atoms differ from sets in that equality/inequality of atoms is determined by a rule special to the type of atoms involved. Note that any object class for which there is given a procedure for determining object equality/inequality can be accommodated without difficulty in a routinely extended set theory.

(c) Any set theory requires an *ordered pair* and *ordered* n-*tuple* notion. In conventional set theory one meets this requirement without introducing objects other than sets by making use of a trick construction. One possible construction of this kind is to define the ordered pair <a,b> as the set {{a},{ϕ,{b}}}, where ϕ denotes the null set. However, in standard set theory, a somewhat different trick is used. One defines the ordered pair <a,b> in terms of unordered pairs as follows:

$$<a,b> = \{\{a\}, \{a,b\}\}$$

Ordered n-tuples are then defined inductively by $<x_1,\ldots,x_n>$ $<<x_1,\ldots,x_{n-1}>,x_n>$. This line of construction, while mathematically neat, leads to programming complications, and we will find it

more convenient to regard n-tuples as non-sets, using conventions whose details will be set forth below.

(d) The *set union* operation is basic to mathematical set theory, and is provided as a primitive in SETL.

(e) The *power set* or *set of all subsets* construction is important in set theory. This is provided in SETL as a basic operation. Note as a *caveat* that for sets as large as the power set of a set having more than a very few elements, efficiency inevitably becomes a serious consideration, and it becomes reasonable to provide as built-in features operations which from the pure mathematical point of view are redundant. As a small concession to this very large fact, we provide in SETL not only the power set construction but an operation which, for a given set, generates the set of all subsets which have a given cardinality.

(f) Set theory makes essential use of the "choice" functions defined by the various versions of the axiom of choice. In SETL, these are all provided by a simple choice operation ∃x, which in an unspecified but implementation-defined way chooses some element out of each non-null set, x.

(g) Another basic primitive of mathematical set theory is the "range of a function" construction. This construction, one of the keystones of set theory in its more elegant representations, asserts that if A(x,y) is any formula of set theory, depending on the two free variables x and y and determining y uniquely for each particular x, then, given any specified set u, there exists a unique set v consisting of all y for which there exists an x in u such that A(x,y). SETL is intended to be executable, and for this reason will not permit a construction quite as general as this; in SETL, we will only allow the range set v to be formed in case we know *a priori* which set w must be searched in order to find the element y whose existence abstract set theory asserts; or if the element y can be constructed in some explicit way from x using basic and programmer-defined SETL operations, so that we may write y = f(x) using a SETL expression f(x) in which the variable x appears. In the former case, SETL will allow us to write v as

$$v = \{y \in w \mid (\exists x \in u)(A(x,y))\} \quad ;$$

in the latter case, as

$$v = \{f(x),\ x \in u\}\ .$$

(h) Set theory assumes at least one infinite set, whose existence is asserted by the normal set-theoretic axiom of infinity. Such a set is used as a starting point to build up the theory of the set of integers (and beyond this, of all of the transfinite cardinals). The sequence of integers is often defined in set theory as ϕ, $\{\phi\}$, $\{\{\phi\}\}$, $\{\{\{\phi\}\}\}$ ... etc. A construction of this kind, while set-theoretically very neat, defines the integers using what is in effect a unary representation, which is of course pointlessly inefficient in view of the existence of binary representations for the integers. In SETL we follow the invariable computer practice and regard the integers as a separate atomic data type, also providing various standard arithmetic operations as basic SETL functions.

In set theory, once the set Z of integers has been defined and the axiom of mathematical induction proved, the existence of Z, and more particularly the possibility of speaking of the whole of Z as a single object, forms the basis upon which one builds all subsequent inductive constructions. To construct a function f(n) defined inductively, the tactic generally used in abstract set theory is as follows: first, using mathematical induction, one proves that for each N $\in$ Z there exists a unique set of ordered pairs, this set constituting the graph of the function f(n) for all n less than or equal to N; then, using the axiom of range, one forms the union of all these sets of ordered pairs, thus obtaining the graph of the entire function f. In SETL we replace this abstract argument by a construct which not only avoids all reference to the infinite totality of integers but has significant advantages of efficiency: namely by programmed iterative loops, and more generally by structured patterns of recursive subroutine calls evaluating an inductively defined function directly. Thus, the label-plus-go-to mechanisms of SETL, and the recursive subroutines which the language provides, are to be regarded as a replacement for the axiom of infinity occurring in abstract set theory. As is known, recursive subroutines alone would suffice for this purpose, but we provide explicitly programmed iterations as well, since they furnish a

mode of epxression more natural in certain circumstances and more efficient in general.

2. Grammar of expressions

The tokens of SETL are names, underlined names, signed integers, real numbers, character string constants, and octal constants which denote bit-strings. Names and signed integers are formed in the usual way; character string constants are included within single quotation marks, and octal constants consist of an octal integer with the suffix B. The formation of character string and octal constants is sufficiently well illustrated by the following examples.

'This is a character string' , 00237B .

(Quotes within quoted character strings are represented by double quotes.) Real numbers are

2.0 , -3.14 , -3.14E-14 .

SETL makes use of the following special symbols, each of which is a lexical delimiter:

( ) [ ] { } , * + - / = ;

< > Ω ∃ ∀ # : . | ↄ ϵ

Besides the operation symbols appearing in this list, additional operation symbols, both system and programmer defined, are provided, and consist of underlined valid names as in eq, ne, with, less, lesf, etc. Names may be used as labels, in which case they are followed by a colon.

The basic entities of SETL are *sets* and *atoms*. Sets have elements and are defined by the elements which they contain. Both sets and atoms may be elements of sets. Atoms are either integers, reals, boolean elements, bit-strings, character-strings, labels, blank atoms, subroutines, functions, or tuples. Expressions of any of the types: tuple, real, set, integer, Boolean, bit-string, character-string, label, blank, subroutine, or function may be written; we shall shortly describe the structure of each of these types of expressions.

2a. Elementary Set Expressions

If x designates a set or atom, while a designates a set, then x ϵ a is a boolean expression having the customary meaning. If a is a name designating any set or atom, then {a} designates the set whose only element is a. More generally, {a,b,...,c} designates the set whose only elements are a,b,...,c. We allow a,b,c, etc., to be arbitrary valid expressions. This bracket construction can be nested to any depth, so that, for example, {a, {a}, {{a}}, {{a},b}} is a valid expression in SETL, a,b designating any sets or atoms. The null set is written as nl.

If a and b are sets, then a + b denotes their union, a * b their intersection, a - b the difference set of a and b, and a/b the symmetric difference of a and b (so that a/b is the same as (a-b) + (b-a)). The expression a + {b} may be abbreviated as a with b; a - {b} may be abbreviated as a less b.

If a is a set, then #a denotes the number of elements of a. Each object in SETL has a type. The type of a set is set.

If a designates a set, then ∋ a designates any arbitrary chosen element of this set. The notation pow(a) designates the set of all subsets of a, and npow(k,a) the set of all subsets of a consisting of precisely k elements. These operations, while provided in SETL, are good examples of the type of set-theoretic construction which should be used very cautiously if impossibly inefficient algorithms are not to result.

2b. Elementary Tuple Expressions.

Besides sets, which are unordered, SETL provides tuples, which are ordered. A tuple may be regarded as corresponding to a sequence $c_1,c_2,c_3,\ldots$ of components, all but a finite number of which are identical with the undefined atom Ω (see below). Two tuples are equal if and only if all their components are equal. The length of a tuple is the index of its last defined component. If this component is c_n, then the tuple whose components are $c_1,c_2,\ldots,c_n,\Omega,\Omega,\Omega,\ldots$ will often be written as

$$\langle c_1,c_2,\ldots,c_n\rangle$$

though of course it might also be written in other ways, as for example

$$\langle c_1, c_2, \ldots, c_n, \Omega \rangle \quad .$$

The tuple described by the sequence of components $\Omega, \Omega, \ldots$, which in the sense defined above is a tuple of length 0, is the null tuple, and may also be written as nult.

The type of a tuple is tupl.

Given a tuple

$$t = \langle c_1, \ldots, c_n \rangle \ ,$$

then

$$t(k)$$

denotes the k-th component of t. The component t(1) may be written as

$$\underline{\text{hd}}\ t \quad .$$

The notation

$$t(i:j)$$

denotes the tuple whose components, for $1 \leq k \leq j$, are t(i+k-1). The notation

$$\#t$$

denotes the length of t. The notation

$$\underline{t\ell}\ t$$

is an abbreviation for

$$t(2:\#t-1) \ ,$$

while t(i:) is an abbreviation for t(i:#t-i+1).

Observe that all of these notational conventions apply also to tuples some of whose components are undefined.

Given two tuples t and t', t+t' denotes their concatenation. E.g. if t = <a,b> and t' = <c,a> then t+t' = <a,b,c,a>.

2c. Functional Application.

If f is a set and a any set or atom, and if f contains precisely one tuple <a,x> whose first component is a then f(a) designates the element x, i.e., the *image* of a under f. More generally, if f is a set of ordered pairs, then f{a} is the set of all x such that <a,x> ∈ f. This is the *set of all images* of a by f. If f{a} contains no element or contains more than one element, then f(a)

the undefined atom Ω. If f and a are sets, then f[a] designates the union of all the sets f{x} for x ϵ a. Note then that f{a} may be written as f[{a}].

More generally and precisely: suppose that f is a set, and x an object other than the undefined atom Ω. Then the elements of the set f{x} are

(a) all the second components y of tuples <x,y> of length 2 belonging to f and having first component equal to x; together with

(b) all the 'tails' $\langle y_1, y_2, \ldots \rangle$ of tuples $\langle x, y_1, y_2, \ldots \rangle$ of length greater than 2 belonging to f and having first component equal to x.

We then define f(x) to be the unique element of f{x}, if this set has a unique element; otherwise to be Ω. The set f[a] is defined as the union of all the sets f{x}, x ranging over all the elements of the set a. (Note that f[a] will also be assigned a meaning for f a character string, bit string, or a tuple; cf. below.)

We also define

f{x,y} to be identical with (f{x}){y},

f{x,y,z} to be identical with ((f{x}){y}){z}, etc.;

and

$$f(x,y,\ldots,z,w)$$

to be identical with

$$(f\{x,y,\ldots,z\})(w) \ .$$

Note that if f is a set of triples <x,y,z>, and if g is the corresponding set of pairs <x,<y,z>>, (which is quite a different set) then f{x,y} and g{x,y} happen to be the same.
It is also convenient to let f{a,[b]} be the union of all the sets f{a,x} for x ϵ b, and more generally to allow such constructions as

$$f\{a_1,\ldots,[a_i],a_{i+1},\ldots,[a_j,a_{j+1},\ldots,a_n],\ldots\} \ ,$$

which denotes the union of all the sets

$$f\{a_1,\ldots,x_i,a_{i+1},\ldots,x_j,x_{j+1},\ldots,x_n,\ldots\}$$

where $x_i \in a_i$, $x_j \in a_j$, $x_{j+1} \in a_{j+1}$, etc.

In all of these expressions we allow f to be not only a function or set name, but any expression whose value is a functio or a set. Thus, for example, the expression

{<a,aa>,<b,bb>,<c,cc>,<a,ax>}{a}

is legitimate, and has the same value as the simpler expression

{aa,ax} .

Of course, if f is a function (which is to say a programmed procedure with a certain definite number of arguments) the correct number of parameters must be supplied to it. Thus, for example, if f is a function of two arguments, the expressions

f(a,b), f[a,b], f{{a},b} ,

etc. are all valid, while

f(a), f[a], f{a,b,[c]}

are all invalid.

2d. Boolean Expressions, Quantified Expressions, Precedence Rules.

Equality, inequality, and set inclusion are provided as operators having the form a eq b, a ne b, a incs b. Bit strings, character strings, labels, blank atoms, subroutines, and functions may be tested for equality and inequality by using the operators eq and ne. The special symbols t, f denote the boolean values "true" and "false" respectively. The standard boolean operations and, or, not, implies, exor are provided; and, not, and implies may also be abbreviated as a, n, imp. Between two integers or reals m and n the usual comparison operators, having the form eq, ne, ge, le, gt, lt are provided. An element may be tested for having "atom" status by applying the operator atom a. (If a is a tuple, atom a is false.)

If e is a set, and c(x) a Boolean formula in which a name x occurs, then a formula of either of the forms

(1) $\exists x \in e \mid C(x)$

(2) $\forall x \in e \mid C(x)$

represents a Boolean value. The value of the first of these expressions, the so-called *existentially quantified* form, is

obtained by calculating the value of the Boolean expression C(x) ›gressively for each of the elements of the set e, and by assigning the value "true" on first obtaining a "true" result, but assigning the value "false" if no such result is obtained. The second of these expressions, the so-called *universally quantified* form, is calculated in corresponding fashion but with "false" replacing "true" and vice versa in the preceding description. In forming expressions like (1) or (2) we wish, however, to exclude such ambiguous cases as

$$\forall x \in e \mid (D(x) \text{ or } \exists x \in e \mid C(x))$$

which are in a sense set-theoretical versions of those ambiguous programming sequences in which an iteration variable is explicitly modified within an iteration, or in which an iteration is headed by some such ambiguous statement, for example the FORTRAN statement

```
DO 1 I=1,I+1 .
```

Such ill-formed cases may be excluded by applying the following technical rule. An occurrence of any name in the role of x in a formula like (1) or (2), i.e., an occurrence of a name x within a part P of a larger formula, P having either the structure $\exists x \mid \text{expr } Q$ or $\forall x \mid \text{expr } Q$, is said to be a *dummy*, or *bound*, *occurrence* of the name x. An occurrence of a name x which is not bound is said to be a *real*, or *free*, *occurrence* of x. We require, in order that the boolean expressions (1) and (2) be properly formed, that all occurrences of the name x in the boolean expression C(x) be free, i.e., that none of these expressions be bound.

Several additional boolean expression forms closely related to the forms (1) and (2) are provided, and have the appearance

(3a) $\min \leq \exists k \leq \max \mid C(k)$ (3b) $\max \geq \exists k \geq \min \mid C(k)$

(4a) $\min \leq \forall k \leq \max \mid C(k)$ (4b) $\max \geq \forall k \geq \min \mid C(k)$

In these formulae C(k) is a boolean expression in which the name k occurs, all of its occurrences being free; min and max are integer expressions in which k does not occur. The value of the first of these expressions is formed by calculating the value of e boolean expressions C(k) for all integers in the range

extending from the value of min to the value of max, assigning the value "true" if C(k) ever assumes the value "true", and assigning the value "false" otherwise. If max < min, (3a) has the value "false". The expression (4a) has an equally evident meaning.

The variant forms (3b) and (4b) provide for a variant order of search. Thus, (3a) implies an iterative search in which integers k are tested in *increasing* order until the expression C(k) takes on the value t; and (4a) an iterative search in which integers k are tested in increasing order until the expression C(k) takes on the value f. Similarly, (3b) implies an iterative search in which integers k are tested in *decreasing* order until the expression C(k) takes on the value t; and (4b) an iterative search in which the integers k are tested in decreasing order until the expression C(k) takes on the value f.

The evaluation of (1) will set x to the first value found for which C(x) is true, if any exists. Similarly, the evaluation of (3a) (resp. (3b)) will set k to the smallest (resp. largest) integer value (in the range shown) for which C(k) is true, if any such value exists. More generally, we allow compound expressions having forms like

$$\exists x_1 \in e_1, \forall x_2 \in e_2(x_1), \forall x_3 \in e_3(x_1,x_2),\ldots|C(x_1,\ldots,x_n)$$

which may be taken in an evident way as abbreviations for expressions compounded using the basic construction (1), (2), (3), (4).

In a boolean expression of this kind, we require that e_1 contain no occurrence of the variables $x_1,\ldots,x_n$; that $e_2(x_1)$ contain no bound occurrence of x_1 and no occurrence of $x_2,\ldots,x_n$, etc. All but the first of a sequence of like quantifiers may be omitted, so that, for example, the formula displayed just above may be abbreviated as

$$\exists x_1 \in e_1, \forall x_2 \in e_2(x_1), x_3 \in e_3(x_1,x_2),\ldots|C(x_1,\ldots,x_n)$$

SETL will use an operator procedure rule which is simpler than those which have become customary. These rules may be

described as follows.

i. All binary operators, except built-in operators producing boolean from non-boolean values, are to have the same precedence. These latter operators, i.e., the operators (ε, eq, ne, incs, ge, le, gt, and lt) have higher precedence, and thus are evaluated first when they occur in compound expressions. Aside from this, unparenthesized expressions containing binary operators of equal precedence are to be evaluated from left to right. Of course, parenthesized expressions will be evaluated in the order indicated by parentheses.

ii. Monadic operators will always have highest prededence, i.e., minimal scope, except as indicated by rule (i). Thus, for example, n x ε a is equivalent to n(x ε a), while n a and b is equivalent to (n a) and b . Note also that - x eq y is - (x eq y), an invalid expression.

iii. As indicated below, SETL allows programmer-defined infix binary and monadic operators. The conventions just described apply to those operators also.

A similar precedence rule is required to avoid ambiguity in the scope of quantifiers. One may ask, for example, if the expression

$$\forall x \in e \mid x \text{ gt } 0 \text{ or } y \text{ eq } 0$$

is to have the reading

$$(\forall x \in e \mid x \text{ gt } 0) \text{ or } (y \text{ eq } 0) ,$$

or the reading

$$\forall x \in e \mid (x \text{ gt } 0 \text{ or } y \text{ eq } 0) .$$

We adopt the following convention. A string of quantifiers terminated by a vertical bar is to be regarded as a monadic operator. As such, it will have minimal scope (except insofar as built-in operators producing boolean from nonboolean quantifiers may have higher precedence). It follows that the first of the above readings is correct.

We note once more that if certain variables $x_1,\ldots,x_k$ occur in a quantified boolean expression B of the above type, and if $,\ldots,x_k$ are all existentially quantified and are not preceded

in the sequence of quantifiers occurring within B by any universally quantified variables, then after evaluation of the expression B, the variables $x_1,\ldots,x_k$ will take on those first values appearing in the iterative search required to evaluate B which yield the value true for B. If no such values exist, then $x_1,\ldots,x_k$ will all take on the undefined value Ω. Thus, for example, if *seq* is a sequence of sets of integers, evaluation of the expression

$$\text{min} \leq \exists k \leq \text{max},\ \forall n \in \text{seq}(k) \mid n \text{ gt } 0$$

will cause k to assume the smallest value, in the indicated range, for which all integers in the set seq(k) are strictly positive. Similarly, evaluations of the expression

$$\text{max} \geq \exists k \geq \text{min},\ \forall n \in \text{seq}(k) \mid n \text{ gt } 0$$

will cause k to assume the largest value, in the indicated range, for which all integers in the set seq(k) are strictly positive. For a third example, note that evaluation of the expression

$$\text{max} \geq \exists k \geq \text{min},\ \exists n \in \text{set}(k) \mid n \text{ gt } 0$$

will cause k to assume the largest value, in the indicated range, for which there exists a positive integer in the set set(k), and will at the same time cause n to assume a positive value belonging to this set; n will be the first positive integer found in the implementation-defined order of search over the set set(k). (If no such n exits, n will be assigned the value Ω.)

2e. Integer Arithmetic Expressions, String Expressions.

The arithmetic operators *, +, -, /, // are provided, the last of these operators denoting the remainder after a division. In addition, the built-in arithmetic operators max, min, abs, the first two of these being dyadic, the last monadic, are provided. Integer arithmetic is carried to as many digits of accuracy as are required for each operation.

The expression m exp n designates m to the n-th power. If n is negative, then the result is real (see the following section)

All the boolean operations apply on a bit-by-bit basis to bit-strings. If two strings of unequal length are combined by

these operations, the shorter is extended by leading zeros to the length of the longer. Boolean quantities are identified with bit-strings of length 1.

Given two strings s and s' of the same type, s+s' designates their concatenation. If n is an integer not exceeding the length of s, then s(n) denotes the n-th bit (or, in the case of a character string, the n-th character) of s. If n is greater than #s, s(n) is Ω. The notation s(m:n) designates that portion of s which starts at its m-th bit (or character) position and extends for n positions. The notation s(m:) designates that portion of s between its m-th position and its last position. The symbols nulb and nulc denote null bit and character strings respectively.

The allowable characters in a character string are all the normal members of the SETL character set, plus one additional character designated er (end record), which plays a special role in connection with input/output (see below).

If n is an integer and s a string, then n * s denotes the result of joining together n identical copies of s end-to-end. If n is an integer, then dec n denotes the (shortest) character string representing n in decimal form, and oct n denotes the (shortest) character string representing n in octal form. If s is a character string representing an integer in decimal (respectively octal) form, then dec s (respectively oct s) yields that integer.

If S is a string and f a function, then f[S] designates the string whose n-th component is f(S(n)).

2f. Real Arithmetic Expressions.

Real arithmetic is provided in SETL, in a manner depending (as usual) on machine and implementation. The arithmetic operations +, -, *, /, and exp (exponential) are provided for real numbers. The operation $real_1$ log $real_2$, where $real_1$ is the logarithmic base, is also provided, as are cos(x), sin(x), x min y, x max y, and abs y. If x is real, top x is the least integer exceeding x, and bot x is the greatest integer not exceeding x. If n is an integer, then top n is the real number most closely approximating n from above, given the limited precision of real numbers. (Note for example that the integer $n=2^{100} - 1$ would be carried precisely in SETL; if hundred bit real arithmetic were being used top n would be the real number 2^{100}.

2g. Label Expressions.

No operations combining labels, except the equality and inequality comparisons, exist; however, labels may be members of sets and may therefore appear within ordered pairs, so that the result of applying a function to an element may be a label. An expression producing a label value may appear in a *go to* statement (see below). This possibility can be used to obtain a "computed" *go to* effect in SETL programs.

2h. Blank Atoms.

Blank atoms are provided for use as structural markers in complex objects built up in the course of a SETL computation; SETL will use blank atoms in many situations in which a pointer-oriented language would use machine addresses or pointers to data blocks.

No operations combining blank atoms, except the equality and inequality comparisons exist; however, blank atoms may be elements of sets, just as atoms of any other kind. Blank atoms are created by the built-in SETL function newat, which creates a new blank atom each time it is called. Note, for example, that such an expression as <newat, newat> designates an ordered pair consisting of *two distinct* blank atoms.

The undefined atom Ω is a particular blank atom related to various SETL operations in somewhat special ways. We do not allow Ω to be a member of any set, so that any attempt to form such a combination as

$$\{\Omega\},\{\Omega,a\}$$

will lead to an error return. However, Ω can be a component of a tuple. An attempt to form any of

$$\exists pow(\Omega),\ \Omega(x),\ f(\Omega),\ f\{\Omega\},\ f[\Omega],\ \Omega,\ \text{or}\ \#\Omega$$

will also lead to an error return. The atom Ω is allowed in the combinations x eq Ω and x ne Ω, but of course not in x gt Ω or in any other arithmetic, string, or boolean operation. (Note that any attempt to execute an operation with arguments of inappropriate type will also lead to an error return.) An

occurrence of Ω either in a quantifier of the form

$$\forall x \in \Omega,\ x \in \Omega,\ \Omega \leq \forall k \leq x, \text{ etc.}$$

in a range restriction

$$x \in \Omega,\ \Omega \leq\ k \leq x, \text{ etc.},$$

or in any iteration control of the form

(while Ω)

will lead to an error return. The value Ω will be returned as the value of ∋ nℓ and as the value of f(a) if a is not in the domain of a set f of ordered pairs. We also have hd Ω =tℓ Ω=Ω.

These conventions help to locate bugs in SETL programs, as they ensure that many situations, in which the actual form of data differs from its assumed form will lead rapidly to the occurrence of Ω as a value and very shortly thereafter to an error return.

2i. Set Formers.

We now wish to describe an important type of set expression having a very great expressive power. This *set former expression* has the following general form

$$(1)\quad \{e(x_1,\ldots,x_n)\ ,\ x_1 \in e_1,\ x_2 \in e_2(x_1),\ldots,\ x_n \in e_n(x_1,\ldots,x_{n-1}) \mid C(x_1,\ldots,x_n)\}$$

In this general expression, $x_1,\ldots,x_n$ are names; e_1 designates a set-expression not containing any occurrence of these names, $e_2(x_1)$ a set expression not containing any occurrence of $x_2,\ldots,x_n$ and containing only free occurrences of x_1, etc. Moreover, $C(x_1,\ldots,x_n)$ designates a boolean expression containing only free occurrences of $x_1,\ldots,x_n$, and $e(x_1,\ldots,x_n)$ designates any arbitrary expression containing only free occurrences of these same names. The notational form (1) is familiar from set theory. Its value is the set formed by the following rule: calculate the set e_1; for each of its elements x_1 calculate the set $e_2(x_1)$; for each of these elements calculate the set $e_3(x_1,x_2)$, etc. For each n-tuple $x_1,\ldots,x_n$ obtained in this way and having the property at the boolean expression $C(x_1,\ldots,x_n)$ has the value "true", calculate $e(x_1,\ldots,x_n)$; gather all these elements into a set,

thus obtaining the value of (1).

The individual restrictions

(2) $$x_j \in e_j(x_1,\ldots,x_{j-1})$$

occurring in (1) may be called *range restrictions*; for use in cases in which ranges of integers play a role, another kind of range restriction, having one of the forms

(3) $\min \leq x \leq \max$, $\min \leq x < \max$, $\max \geq x \geq \min$, etc.

is provided. (cf. Section 2d). If a range restriction of the type of (3) occurs in the formula (1) in place of one of the range restrictions (2), then, for each appropriate $x_1,\ldots,x_{j-1}$ the two arithmetic expressions $\min_j(x_1,\ldots,x_{j-1})$ and $\max_j(x_1,\ldots,x_{j-1})$ will be calculated, and in the formation of the set (1) x_j will vary over the numerical range determined by these upper and lower limits (in the indicated order).

If no particular boolean condition C is to be imposed, then the expression (1) may be written as

$$\{e(x_1,\ldots,x_n),\ x_1 \in e_1,\ x_2 \in e_2(x_1),\ \ldots,\ x_n \in e_n(x_1,\ldots,x_{n-1})\}\ .$$

The special case

$$\{x,\ x \in e \mid C(x)\}$$

may be abbreviated as

$$\{x \in e \mid C(x)\}\ .$$

2j. Conditional Expressions.

A conditional expression like that of ALGOL is provided; this has the form

if $bool_1$ then $expr_1$ else if $bool_2$ then $expr_2$ ... else $expr_n$

and has its customary and evident meaning. Here, $bool_1,\ldots,bool_{n-1}$ are required to be boolean expressions, while $expr_1,\ldots,expr_n$ are expressions having arbitrary values.

A resolving convention is required if the implied scope of the concluding *else* in a conditional expression is not to be ambiguous. One may ask, for example, if the expression

(1) if x gt 0 then y else x + y

to have the reading

(2) if x gt 0 then y else (x+y)

or the reading

(3) (if x gt 0 then y else x) + y .

We resolve this ambiguity by agreeing that the concluding *else* in a conditional expressions is to be regarded as a monadic operator. As such, it will have minimal scope (except insofar as built-in operators producing boolean from nonboolean quantities may have higher precedence). It follows that the normal reading of (1) is (3).

2k. The Use of Functions within Expressions; Programmer-Defined Operations.

SETL allows subroutines and functions to be defined in a manner to be described in more detail below. Functions may be used as parts of expressions; a function invocation occurring within an expression will have the form

name(exprl,...,exprn) .

Here, name is the function name, while exprl,...,exprn may be arbitrary SETL expressions. These expressions are evaluated before the function is invoked, and the values thereby obtained define the arguments to be transmitted to the function. A function called from within an expression may modify various of the function's arguments, and its invocation may in general cause other "side effects". Function arguments whose values are to be modified should be simple names rather than compound expressions.

Subroutines and functions may be the values of expressions. Subroutines and functions may be compared for equality and inequality. They may also be applied to other elements, either in the form f(x), or in the form f[x], or in any of the related, more general, forms that have been described in section 2c. Note, for example, that if x is a set, y an integer, and g(z) a function-valued function whose value is F, then g(z){[x],y} is a valid

expression, having the same value as F{[x],y}. That is, its value is the set consisting of all the elements F(u,y), where u ∈ x.

The value of a function f of zero arguments is written as f( ).

SETL allows programmer-defined functions and subroutines of two arguments to be written as infix operators, provided that this notational intent is appropriately stated in the definition of the function (additional details are given below). Similarly, functions of a single argument may be written as prefixed monadic operators. The name of a function to be written in such an operator form, whether monadic or dyadic, is underlined. Thus, for example, if reverse and invert are the names of programmer-defined dyadic operators we may write

(seta reverse setb) invert setc

as part of a compound expression.

The basic SETL expression forms may be nested in arbitrary fashion to produce complex expressions. Thus, set-forming constructions may be nested within conditional expressions, which may in turn be nested within n-tuple forming expressions, etc. The order of expression evaluation is inside-out and left-to-right. This observation concerning evaluation order may be important if functions producing side effects are invoked during the evaluation of an expression. Function evaluation is in systematic inner-to-outer, left-to-right order.

2ℓ. Examples of the Use of the SETL Expression Forms.

It is not uncommon for operations which would have to be represented by short programs in lower-level languages to reduce to simple expression evaluations in SETL. Often search loops are replaceable by existential expressions, procedures which build up arrays by set-formers, and so forth. Numerous substantial programs exemplifying this remark will be given in later portions of the present manuscript. Here we give only a few simple examples. In this example we use the SETL print statement.

This statement will be described in more detail in a later section. or the present, it is sufficient to remark that

print e;

prints the value of the expression e using standardized formatting conventions.

Example 1: Print the set of all primes not exceeding n.

```
print {1 < m ≤ n | (not 1 < ∃k < m | (m // k) eq 0)};
```

Example 2: Print the first nonempty string of symbols which occurs with an asterisk immediately to its left and right within a string s. Print a diagnostic message if there is no such substring.

```
print if not 1 ≤ ∃m ≤ #s, m+1 < n ≤ #s|s(m) eq '*' and s(n) eq '*'
   then 'no substring of type sought'
   else s(m+1: n-m-1);
```

Example 3: Form and print a mapping giving the number of times each character appears in a character string s.

```
print {<c, #{1 ≤ n ≤ #s | s(n) eq c}>,
                          c ∈ {s(n), 1 ≤ n ≤ #s}};
```

2m. The object-type operator. The special operator 'is'.

Occasionally one will want to test the type (i.e., set, tuple, string, etc.) of a SETL object, or to use this type as the basis for a go-to with calculated label. Accordingly, the operator type is provided, type x being the type of x. The value of type x is a blank atom, more specifically one of the blank atoms designated by the special constant symbols integer, real, cstring, bstring, subroutine, function, label, blank, tupl, set. Thus, for example, the test for a pair is

(1) `if (type x) ne tupl then f else (# x) eq 2 .`

The special binary operator is provides a convenient on-the-fly assignment form like that available in APL. The expression

) `x is y`

has x as its value, but when evaluated makes the value of y (which must be a simple variable) equal to that of x. In (2), x can be any expression; (2) itself is an expression and can be used as part of a compound expression.

The is operator obeys somewhat nonstandard precedence conventions. It has minimal left-hand and maximal right-hand precedences. Thus, for example,

(3) x+y is z+w

has the significance

((x+y) is z)+ w

(and the value (x+y)+w); even

(4) u+v is f(w)

has the significance

(u+v is f)(w)

and therefore the same value as

(u+v)(w)

(which is of course a functional evaluation or indexing operation).

Such combinations as

u+v is (x+y)

are illegal.

Item 13. DESCRIPTION OF THE SETL LANGUAGE.

Second Part: Assignment Statements.

SETL admits various expression forms in addition to those described above. Many of these additional expression forms have a syntax and semantics which relates them closely to some of the statement forms used in SETL. For this reason, we postpone the discussion of these expression forms, and proceed now to discuss the statement forms of the SETL language, starting with assignment statements.

Note to begin with that single statements of the SETL language are punctuated with terminating semicolons.

Note also that SETL provides for the use of comments, which, being entirely ignored by the SETL processor, can be inserted anywhere in a SETL text except within a single token. As in PL/1, comments are enclosed fore-and-aft by use of the composite marks /* (prefixed) and */ (affixed). Thus, for example,

```
/* this is an example of a setl comment. */ .
```

The simplest kind of assignment statement has the familiar form

```
name = expr;
```

here *name* must be any valid variable name, while *expr* can be any valid expression.

In addition to this simple form of assignment statement, we also wish to make use of various assignment forms related to the "indexed" assignment operations conventionally used in programming languages. The simplest such operator would have the form

```
(1)                name(expr1) = expr2;
```

The semantics of the statement (1) should be such as to force the expression name(expr1), when evaluated subsequently, to yield expr2 as its value. Thus, if the value of *name* is a set, then (1) will have the same force as

```
name = {p ∈ name|if type p ne tup1 then t else p(1) ne(expr1)}
                    with <expr1,expr2> .
```

Similarly, we will wish to make use of such assignments as

(2) $\underline{hd}\ x = a \quad \text{and} \quad \underline{t\ell}\ y = b$.

The semantics of (2) must be such as to force the expressions $\underline{hd}\ x$ and $\underline{t\ell}\ y$, when subsequently evaluated, to yield a and b as their respective values. It is therefore reasonable to interpret the first of the assignments (2) as being equivalent to $x = \langle a \rangle + \underline{t\ell}\ x$; and the second assignment as being equivalent to $y = \langle \underline{hd}\ y \rangle + b$. Thus we see that operators of various kinds can be expected to appear on the left-hand side of SETL assignment statements. The power and flexibility of assignment statements in SETL will be greatly increased by allowing operators appearing in this "sinister" position (i.e., on the left-hand side of assignment statements) to be compounded. This possibility has not been regularly exploited in programming languages, for which reason we shall explain both the syntax and semantics of our intended constructions in more detail than would otherwise be necessary.

The scheme proposed is generally applicable to "procedural" programming languages. It is based upon certain general algebraic observations concerning "retrieval and assignment" pairs of functions which will be presented below. The scheme avoids the explicit transmission of pointers, and the complications which such transmission may lead to. The mechanisms provided constitute systematic generalization to the left-hand sides of assignments of the standard subroutine-function linkage concepts applying to the right-hand side of an assignment; and are as basic as (though not necessarily as generally useful as) these latter concepts. For this reason, we shall call the interprocedural linkages to be suggested "sinister calls", and call the conventional call linkages used on the right-hand side of an assignment "dexter calls".

To stress the general nature of the considerations involved, we shall begin with a few very general remarks. SETL belongs to the class of "procedural" languages, that is, to those languages which in one or another manner represent the algebra of transformations on a universe of stored data objects. The semantic operations most fundamental to such languages are:

i - access to a designated portion of a stored object.

ii - modification of a designated portion of a stored object.

iii - combination of transformations by successive application.

iv - choice of transformation to be applied depending on a predicate of particular stored data objects.

v - combination of two or more stored objects in some "algebraic" fashion, useful generally or for some specifically intended application area, the outcome of this combination process being some "output" object.

vi - repetitive application of a transformation until a certain condition is established.

With these primary operations various secondary operations may be listed; of these possibilities we shall mention only

vii - application of a given transformation to all the subobjects of a given object (iteration-over-parts).

All the considerations in the next few pages will center around operations i and ii. We call these operations retrieval and storage operations respectively.

Let us now proceed with a more detailed discussion, starting with an example, which will serve to fix our overall goal firmly in mind.

Using the scheme to be suggested in the pages which follow it will be possible to program a function called last, which when called in the normal dexter way returns the last element of an n-tuple; and then to use this function to the left of an equal sign, writing

last tupl = x

to change the final (and only the final) component of *tupl*. Then, for example, by executing

```
x = <<1,2,3>,4,5>;
last hd x = 10;
```

one will cause x to take on the value

<<1,2,10>,4,5>,

etc.

The key to the situation that is to be studied lies in the proper definition of storage operators and retrieval operators. Under what conditions will we wish to say that an operator has one of these two

characteristics? First consider retrieval; and let op stand for an operator. If this is to be a retrieval operator, it must in the first place be free of "side effects". That is, if op a is called once, and then again, the same value should be returned both times. More generally, if t is a temporary variable not occurring elsewhere in a program, then execution t = op a; should have no influence on the remainder of the program.

Next there should exist a storage procedure corresponding to the retrieval operator. This will be some procedure, call it

(1) opstore(a,x);

such that after executing (1) we can be sure that an immediately following call to op a will return the value x. Moreover, two successive calls to (1) should have the same effect as a single call; and, more precisely, whenever the value of op a is already x, opstore(a,x) should be an identity transformation. The last requirement is rather sharp, and pins opstore(a,x) rather closely to op a; moreover, it implies that the property "to be a retrieval operator" is not possessed by all operators op.

Note the following simple retrieval storage operation pairs in SETL: (some of these pairs stand in the retrieval-storage relationship only in most cases, but not in all possible degenerate cases):

(2a) *retrieval operator:* hd a
storage operator: a = <x> + tl a;

(2b) *retrieval operator:* tl a
storage operator: a = <hd a> + x;

(2c) *retrieval operator:* a(z) (for a set a containing n-tuples)
storage operator: a = {y ∈ a|if (type y) ne tupl then t
else(n(hd y)eq z)or(#y)le 1}with <z,x>;

(2d) *retrieval operator:* a{z}
storage operator: a = {y ∈ a|if (type y) ne tupl then t
else(n(hd y) eq z)(#y) le 1}
+ {<z,w>, w ∈ x};

and among a large family of other more complex examples which might be cited

(2e) *retrieval operator:* a(i:j) (for a tuple or string a)
storage operator: a = a(1:i-1) + x + a(j+1:#a-j) .

Suppose that op is a retrieval operator for which some associated storage procedure opstore(a,x) has been selected. We will find it heuristically and syntactically convenient to indicate a call on this storage procedure by writing

op a = x;

That is, we indicate a call on the storage procedure by writing the retrieval operator in a syntactically 'sinister' position.

By defining a formal notation of independent storage operators we help ourselves to pin down the intuitive idea that a storage operator should change no more than is required by its relation to a particular retrieval operator:

Let op and op' be two retrieval operators; anticipating the syntactic style to be developed we shall write calls on their associated storage procedures as

(3a) op a = x; and (3b) op' a = x;

respectively. We call op and op' independent retrieval operators if in the succession of calls

(4) y = op a; op' a = x; z = op a;

the variables y and z necessarily receive the same value; provided that the same is true when op' and op exchange roles.

The following observation is now crucial: Let two retrieval operators op and op', not necessarily independent, be given. The composite operator op op' is a retrieval operator; we can define its storage routine by the code sequence

```
 i.   t = op' a;  /* where 't' is a 'compiler temporary' */
 ii.  op t = x;
 iii. op' a = t;
```

the effect of this may appropriately be represented by writing

(5) op op' a = x;

Proof: (which please ponder). (1) In the first place, we must show that op op' has no side effects. But if t is a temporary variable t occurring elsewhere in a program, then execution of t =op op' a; i.e., of t_1 = op' a; t = op t_1; where t_1 is another such temporary variable clearly does not influence the remainder of the program.

(2) Next we show that the second of two immediatively successive executions of the storage sequence (i,ii,iii) is equivalent to a no-operation. Specifically, suppose that we execute:

$$
\begin{aligned}
t &= \underline{op}'\ a;\\
\underline{op}\ t &= x;\\
\underline{op}'\ a &= t;\\
t_1 &= \underline{op}'\ a;\\
\underline{op}\ t_1 &= x;\\
\underline{op}'\ a &= t_1;
\end{aligned}
$$

Then the third and fourth operations are clearly equivalent to $t_1 = t$; and thus the sequence as a whole is equivalent to

$$
\begin{aligned}
t_1 &= \underline{op}'\ a;\\
\underline{op}\ t_1 &= x;\\
\underline{op}\ t_1 &= x;\\
\underline{op}'\ a &= t_1;
\end{aligned}
$$

Since the second of two identical successive stores is equivalent to a no-op, this last code sequence is plainly equivalent to the sequence i, ii, iii.

(3) Next suppose that we perform the storage sequence (i,ii,iii) and then immediately perform the retrieval u = $\underline{op}\ \underline{op}'$ a; i.e., $t_1 = \underline{op}'$ a; u = $\underline{op}\ t_1$; . We must then show that the retrieval can be replaced by u = x;. But, because of (iii), $t_1 = \underline{op}'$ a; is equivalent to t_1 = t; so that because of ii the whole sequence is equivalent to (i,ii,iii) followed by u = x; .

(4) Finally we consider the case in which the storage sequence (i,ii,iii) is preceded by the retrieval x = $\underline{op}\ \underline{op}'$ a; . We must then show that the storage sequence is equivalent to a no-operation.

If the retrieval ($\underline{op}\ \underline{op}'$ a) already gives x, then after i the retrieval ($\underline{op}$ t) gives x also. Hence the operation ii may be omitted. But therefore the retrieval ($\underline{op}'$ a) gives the value t; and hence iii may be omitted. Therefore the whole sequence i,ii,iii amounts merely to a store into a compiler-generated temporary; and is therefore equivalent to a no-operation. Q.E.D.

The construction i,ii,iii may evidently be iterated, with the following result. Let $\underline{op}$, $\underline{op}'$, ..., $\underline{op}^{(n)}$ be a sequence of retrieval operators. Then their product $\underline{op},\underline{op}',\ldots,\underline{op}^{(n)}$ is also a retrieval operator, and the assignment operation:

(6) $\underline{op}\ \underline{op}'\ \ldots\ \underline{op}^{(n)}\ a = x$

xpands naturally as

(7)
$$\begin{aligned} t^{(n)} &= \underline{op}^{(n)}\ a; \\ t^{(n-1)} &= \underline{op}^{(n-1)}\ t^{(n)}; \\ &\vdots \\ t' &= \underline{op}'\ t"; \\ \underline{op}\ t' &= x; \\ \underline{op}'\ t" &= t'; \\ &\vdots \\ op^{(n)}\ a &= t^{(n)}; \end{aligned}$$

Note that if $x = \underline{op}\ \ldots\ \underline{op}^{(n)}\ a$, then the middle and all following lines in (7) are equivalent to no-ops.

We shall speak of the code sequence (7) implied by the statement (6) as unraveling (6).

It is worth noting one additional feature of the expansion (7). Suppose that $\widehat{\underline{op}}^{(n)}$ is a retrieval operator independent of $\underline{op}$, and that $\widehat{\underline{op}},\ldots,\widehat{\underline{op}}^{(n-1)}$ is any sequence of retrieval operators. Then the products

(8) $\widehat{\underline{op}}\ \widehat{\underline{op}}'\ \ldots\ \widehat{\underline{op}}^{(n-1)}\ \widehat{\underline{op}}^{(n)}$ and $\widehat{\underline{op}}\ \widehat{\underline{op}}'\ \ldots\ \widehat{\underline{op}}^{(n-1)}\ \underline{op}$

are also independent. Indeed, in this case, the only assignment to $\underline{a}$ in (7), that is, the last line in (7), has no effect on the value of $\widehat{op}^{(n)}\ a$, and thus none on the value of

$$\widehat{\underline{op}}\ \ldots\ \widehat{\underline{op}}^{(n)}\ a\ .$$

More generally, if $\widehat{\underline{op}}^{(j)}$ is independent of $\underline{op}^{(j)}$, then

$\widehat{op}\ldots\widehat{op}^{(j-1)}\widehat{op}^{(j)}op^{(j+1)}\ldots\widehat{op}^{(n)}$ and $\widehat{op}\ldots\widehat{op}^{(j-1)}op^{(j)}\ldots\widehat{op}^{(n)}$

are independent. This conclusion may be proved by "algebraic" reasoning from the previous special case, and may also be demonstrated directly.

Our conclusions up to this point may be summarized in the following

Statement A: *The set of retrieval operators associated with the set of stored objects of a procedural programming language forms a semigroup, associated with the language in a natural way.*

The following operations are basic retrievals in SETL:

f(a), f{a}, hd f, tl f, f(i:j)

The storage operators associated with these retrievals have been displayed above. Algebraic relationships between these basic retrievals and other more compound operators lead in accordance with the preceding discussion to other retrieval operators, and to significant relations between storage procedures. Note for example that ∋a is a retrieval operation, the corresponding storage procedure being a = {x}; . The expression f(a) is, if defined, logically identical with the compound ∋f{a}. The reader can verify that the basic definition i,ii,iii associates with this compound retrieval precisely the storage procedure (2c). The operation f{a,b} has been defined in section 6.2(c) as being equivalent to the compound (f{a}){b}. The reader may verify that the general rule i,ii,iii associate with this retrieval the storage procedure

f = {y ∈ f|if(type y) ne tupl then t else if(type tℓ y) ne tupl
then t else (hd y) ne a or (hd tℓ y) ne b or (#y) le 2}
+ {<a,<b,z>>, z ∈ x};

which in accordance with our general notational conventions we can write as

$$f\{a,b\} = z;$$

The same line of argument leads us to define the general assignment

$$f\{a_1,\ldots,a_n\} = x;$$

as having the significance

f = {y ∈ f|if(type y) ne tupl then t else if(type tℓ y) ne tupl
then t else if(type tℓ tℓ y) ne tupl then t else ... else
(hd y) ne a_1 or (hd tℓ y) ne a_2 or ... or(#y) le n}
+ {$<a_1,<a_2,\ldots,<a_n,z>\ldots>>$, z ∈ x};

In the same way, we are led to give

$$f(a_1,\ldots,a_n) = x;$$

the significance

f = {y ϵ f|if type y ne tupl then t else if(type tℓ y) ne tupl
then t else if type tℓ tℓ y ne tupl then t else ... else
(hd y) ne a_1 or (hd tℓ y) ne a_2 or ...}
with $\langle a_1,\langle a_2,\ldots,\langle a_n,z\rangle\ldots\rangle\rangle$;

In accordance with our general notational conventions, we will also write calls on the storage procedure shown in (2e) as

a(i:j) = x; .

(Note that (2e) only defines a storage procedure if (#x) eq y.)

The range operation f[a] may also be regarded as a retrieval operation, though the corresponding storage procedure is highly nonunique. Somewhat arbitrarily, we take

f[a] = x;

to mean

f = {y ϵ f|if hd y eq Ω then t else if (hd y)n ϵ a then t else f}
+ {<y,z>, y ϵ a, z ϵ x};

Similarly,

$f[a_1,\ldots,a_n]$ = x;

will mean

f = {y ϵ f|if hd y eq Ω then t else if hd y n ϵ a_1 then t
else if hd tℓ y eq Ω then t else if hd tℓ y n ϵ a_2 then t
... else f} + $\{\langle y_1,\langle y_2,\ldots,\langle y_n,z\rangle\ldots\rangle\rangle$,
y_1 ϵ a_1, y_2 ϵ a_2, ..., y_n ϵ a_n, z ϵ x};

We take the special form

f(y) = Ω;

to have precisely the same significance as

f{y} = nℓ; .

More generally, we take

$f(y_1,\ldots,y_n)$ = Ω;

have the same significance as

$f\{y_1,\ldots,y_n\}$ = nℓ; .

Concluding this digression on various important particular storage operators, we return to a more theoretical discussion of retrieval and storage operators, and note that we may also consider retrieval operators $op(p_1,\ldots,p_n,a)$ depending on several parameters, and the storage subroutines $opstore(p_1,\ldots,p_n,a,x)$ associated with them. We require that if the function $op(p_1,\ldots,p_n,a)$ has the value x, then the subroutine $opstore(p_1,\ldots,p_n,a,x)$ acts as an identity operator. In this case, a call to opstore may be written in the form

(9) $$op(p_1,\ldots,p_n,a) = x;$$

though other syntactic forms (both for the left- and the right-hand sides) might be preferred in particular cases.

As with simple retrieval operators, so in the case of retrieval operators with parameters the composition of two retrieval operators is a retrieval operator. The natural interpretation of

(10) $$op(p,op'(q,a)) = x$$

is

(11)
```
i'.   t  =  op'(q,a);
ii'.  op(p,t)  =  x;
iii'. op'(q,a) = t;
```

cf. i,ii,iii and (7). Note in particular that if

(12) $$x = op(p,op'(q,a)) \ ,$$

then after (11.i') is executed $op(p,t)$ has the value x and $op'(q,a)$ has the value t; hence (11.ii') and (11.iii') may be omitted. This is to say that the whole sequence (11) reduces to a no-operation.

Having come this far, we may now observe that close examination of (11) reveals a fact which permits very extensive generalization of the "unraveling" constructions i,ii,iii and i',ii',iii'. Namely, we see that the unraveling (11) of (10) treats all the arguments of *op* and *op'* on an equal footing, making it unnecessary to distinguish between a single "principal argument" and a remaining set of

"parameters". To emphasize this point, it is well to study an example. Consider a hypothetical three-parameter function *select(f,g,j)* which returns the value defined by the expression

(13) if j gt 0 then hd f else tl g

This is a multiparameter retrieval operation, possessing an associated storage procedure

(14) if j gt 0 then hd f = x; else tl g = x;;

Having made this observation, we may observe that (11) automatically assigns a meaning to such a statement as

(15) select (f, select(g,h,i), j) = x;

Indeed, the reader aware that the standard nested (dexter) function calls assigns the value

(16) if j gt 0 then hd f else if i gt 0 then tl hd g else tl tl h

to the expression

(17) select(f, select(g,h,i), j)

will verify without difficulty that if (15) is unraveled in accordance with the general convention (11) there results a procedure equivalent to the conditional statement

(18)
```
if j gt 0 then hd f = x;
   else if i gt 0 then tl hd g = x;
        else tl tl h = x;;
```

It may also be remarked that the 'unraveling' process discussed above may be carried over to more general nests of sinister calls. Consider, for example, the retrieval function *select* described by (13) (and (14)) above. It is heuristically clear that one ought to be able to assign a reasonable meaning to

(19) select(select(f,g,i),select(ff,gg,i),j) = x; .

If the compound form appearing on the left appeared on the right instead, it would retrieve

```
if j gt 0 then if i gt 0 then hd hd f else hd tl g
          else if i gt 0 then tl hd ff else tl tl gg
```

making it plain what storage operation (19) ought to represent. The appropriate way to unravel (19) is as follows:

(20) i. t = select(f,g,i);
ii. tt = select(ff,gg,i);
iii. select(t,tt,j) = x;
iv. select(f,g,i) = t;
v. select(ff,gg,i) = tt; .

Note now that the sequence (20) is appropriately related to the dexter call

(21) x = select(select(f,g,i), select(ff,gg,i),j);

Indeed, if (21) is executed immediately before (20), then after (20.i) and (20.ii) have been executed, we have

(22) select(t,tt,j) ≡ x ,

so that (20.iii) is equivalent to a no-op, and may be removed. But then (20.iv) is a no-op, since preceded by (20.i); and (20.v) a no-op, since preceded by (20.ii). The storage-retrieval relationship between (20) and (21) is therefore plain.

The formal argument just given plainly applies to arbitrary combinations of retrieval functions by nesting; this remark leads to the following substantial generalization of the fundamental statement A made above.

Statement B: *The family of multi-parameter retrieval operators associated with the set of stored objects of a procedural programming language is closed under the operation of substitution.*

Yet another property of our procedure for unraveling a nested sinister call is worth noting. If we consider the sinister call

(23) select(f,g,select(ff,gg,i)) = x;

and note from the definition (13) of *select* that this function does not modify its third argument, it is apparent that the most appropriate expansion of (25) is

(24) t = select(ff,gg,i);
select(f,g,t) = x; .

That is, one would want to regard the inner call to *select* as being implicitly dexter. Our normal sinister call expansion, applied mechanically, would instead give

(25)
```
i.    t = select(ff,gg,i);
ii.   select(f,g,t) = x;
iii.  select(ff,gg,i) = t;
```

But (24) and (25) are equivalent! Indeed, since (25.ii) does not change t, it follows that (25.i) and (25.iii) remain mutually inverse retrieval and storage calls, so that (25.iii) is a no-operation. Aside therefore from implications concerning efficiency, the standard sinister expansion (25) is perfectly acceptable. Note also that an optimizer capable of detecting the fact that select does not vary its left-hand side could automatically exploit this fact to suppress (25.iii) as redundant.

The procedure for expanding sinister calls suggested by (6)-(7) and (19)-(20) is thus general and unambiguous.

The reader will perceive that the conventions we have introduced allow wide generalization of the forms which can appear on the left-hand side of an assignment statement. In a subsequent section we will extend this generalization still further, developing mechanisms which allow arbitrary programmer-defined functions to appear in sinister position. The details of this final generalization will, however, be understood best after the syntactic conventions used in connection with function definitions and function calls have been explained. For this reason, we postpone discussion of the use of general programmer defined functions on the left-hand side of assignment statements, and confine ourselves at the present time to discussing the manner in which forms compounded of SETL primitives can be used in sinister position. We have already noted that SETL provides.

(26) $f(a_1)=x$; $f(a_1,\ldots,a_n)=x$; $f\{a_1,\ldots,a_n\}=x$; and $f[a_1,\ldots,a_n]=x$;

as sinister forms, and provides the forms

(27)
```
hd f = x;        tl f = x;    f(i:j) = x;
```

well. We use the same conventions for operation compounding on the left as on the right-hand side of an assignment statement. Thus

(28) hd tl(f(i:j)) = x; (f(a)){b_1,b_2} = x;

are valid also. Note once again that the meaning of compound sinister forms like (28) is in each case to be deduced from the meaning of the corresponding primitive assignment (26) or (27) by using expansion rules like (6)-(7) or (10)-(11).

Next note that the tuple former <a_1,...,a_n> may be regarded as a multi-parameter retrieval operator, its associated storage procedure being defined by the sequence of statements

(29) a_1=x(1); ..., a_{n-1}=x(n-1); a_n=x(n);

Accordingly, we allow "multiple assignment statements" of the form

(30) <a_1,a_2,...,a_n> = x;

we define the significance of (30) by (29). In accordance with the general expansion rules applying to compound forms this basic definition also assigns the significance of such compound forms as

(31) <<a,b>,c,<d,e>> = x;

Moreover, forms such as

(32) <f(a),g(b),h(c)> = x;

also are valid. Note also that forms like

(33) <a,f(a)> = x;

are handled unambiguously by our expansion rules. For example, (33) expands as

(34)
```
i.    t = f(a);
ii.   <a,t> = x;
iii.  f(a) = t;
```

which since (ii) completely reassigns its second argument has precisely the same effect as

(35) a = x(1); f(a) = x(2);

SETL conditional expressions (cf. section 6.2j) can also be used on the left-hand side of assignment statements. Thus, for example, we allow

```
if j gt 0 then hd f else tl g = x;
```

...is has the same force as the conditional statement

```
if j gt 0 then hd f = x; else tl g = x;;
```

(Additional details concerning conditional statements in SETL are given below).

Note finally that in offering an abstract definition of the storage-retrieval relationship we proceed along a line of thought familiar to the hardware designer. At the hardware level, 'storage' and 'retrieval' may involve extensive recoding, complex transformations and reshufflings, etc. However, since the abstract storage-retrieval relationship is always maintained, none of this affects the programmer's fixed picture of the basic logic of these operations. The sinister call mechanism which has just been outlined makes a similar facility available at the programming-language level. It deserves to be emphasized that the use of this facility can decouple a good part of the logic of algorithms from very extensive 'behind the scenes' operations set in motion by storage or retrieval requests; in the same way, the hardware designer decouples the programmer from the details of his 'paging' operations. Thus other generalized assignment notions which have been presented isolate the problem of memory-milieu definition from the remainder of a complex programming task, and keeps globally used data objects from propagating complexity in as virulent a fashion as would otherwise be the case.

Item 14. DESCRIPTION OF THE SETL LANGUAGE.

Third Part: Additional Statement and Expression Forms.

Having described the SETL assignment statement in a manner sufficiently general for the moment, we now go on to describe various other statement forms provided in SETL.

1. Labels, Go-to Statements, Iterations, and Compound Operators.

Control of program flow is provided in several manners; by *go to* statements, by *if* statements, by *flow* statements, by a statement form specifying iteration over a range of sets, by a statement form specifying iteration for as long as a certain boolean condition is valid; and by subroutine or function calls. Because they deserve separate discussion, the *if* and *flow* statements will be discussed in Section 3.

A SETL statement may be labeled by prefixing it with a name, which must be followed by a colon, and which for the sake of readability may be enclosed in pointed brackets. The affixed colon designates the name as a label. Thus

(1) label: , <label:> , <<label:>>

are all equivalent valid labels.

A *go to* statement has the form

go to expr;

the expression occurring in such a statement may be perfectly general, but must have a label as its value.

SETL allows iterations to be specified without the explicit use of labels. Several statement forms serving this purpose are provided The first, which may be called the set-theoretic iteration, has the general appearance

(2) $(\forall x_1 \in e_1,\ x_2 \in e_2(x_1),\ldots,x_n \in e_n(x_1,\ldots,x_{n-1}) \mid C(x_1,\ldots,x_n))$block;

In this general expression, $x_1,\ldots,x_n$ are names, e_1 designates a set-expression not containing any occurrence of these names,

$e_2(x_1)$ a set expression not containing any occurrence of $x_2,\ldots,x_n$ and containing only free occurrences of x_1, etc. Moreover, $C(x_1,\ldots,x_n)$ designates a boolean expression containing only free occurrences of $x_1,\ldots,x_n$, while *block* is any sequence S of valid SETL statements and may include *go to* statements. The statement (2) is executed according to the following rule: calculate the set e_1; for each of its elements x_1 calculate the set $e_2(x_1)$; for each of these elements, calculate the set $e_3(x_1,x_2)$, etc. For each n-tuple $x_1,\ldots,x_n$ obtained in this way and having the property that the boolean expression $C(x_1,\ldots,x_n)$ has the expression *true*, perform the statements of the sequence S in order. (Note that the occurrence in S of a go to statement with a destination label outside S will terminate the iteration implied by the statement (2).)

The individual restrictions

(3) $$x_j \in e_j(x_1,\ldots,x_{j-1})$$

occurring in (2) may be called *range restrictions*; for use when iteration over a range for integers is desired, a restriction having the variant form

(4A) $$\min_j(x_1,\ldots,x_{j-1}) \leq x_j \leq \max_j(x_1,\ldots,x_{j-1})$$

is provided. If (4A) occurs instead of (3) in an iteration (2), then for each appropriate $x_1,\ldots,x_{j-1}$ the two arithmetic expressions $\min_j(x_1,\ldots,x_{j-1})$ and $\max_j(x_1,\ldots,x_{j-1})$ will be calculated, and iteration will be extended suitably over all x_j in the numerical range defined by these upper and lower limits.

Numerical range restrictions in the variant forms

(4B) $$\max_j(x_1,\ldots,x_{j-1}) \geq x_j \geq \min_j(x_1,\ldots,x_{j-1})$$

(4C) $$\min_j(x_1,\ldots,x_{j-1}) \leq x_j < \max_j(x_1,\ldots,x_{j-1}) \text{ , etc.}$$

are also allowed. These variant forms provide for variant orders of iteration. Thus, (4A) implies an iteration in which successive integers x_j are treated in increasing order; (4B) implies an iteration in which successive integers x_j are treated in decreasing order.

Iteration over an empty set is allowed, in which case the block statements in the scope of the iteration is not executed.

The scope of a set-theoretic iteration of the form (2) may be indicated, in the manner shown, by the presence of a semicolon otherwise absent. For readability, however, several alternative forms are provided. These are:

1. The use of the terminator

end;

2. The use of the terminator

end $\forall t_1\ t_2 \ldots t_k$;

to close the scope of an iteration which begins

$(\forall t_1 \ldots)$.

Here, $t_1, t_2, \ldots, t_k$ designate the first k tokens following the iteration-opening symbol $\forall$.

A second type of iteration, which may be called the *while*-iteration, is provided in two related forms, of which the simpler is

(5) (while C) block;

Here C is any boolean expression, while *block* designates any sequence S of statements. This statement performs *block* iteratively as long as C has the value true, but terminates as soon as C is found to have the value false.

The outer extent of a while-iteration's scope may be indicated, as in (5), by the presence of a semicolon otherwise absent. For readability, however, several alternative forms are provided. These are the more visible terminators

end; or end while;

The latter terminator may optionally be made more explicit by extending it to include a number of tokens following the keyword *while* which opens the iteration. This possibility is illustrated in the following example.

(while x ∈ s) k = k+g(x); x = f(x); end while x; .

A rather more general form of while-iteration is as follows:

(while C doing blocka) blockb; .

Here C is any boolean expression, while blocka and blockb are arbitrary sequences of statements. This iteration has precisely the same significance as does the following simple while-iteration:

(6) (while C) blockb blocka;

This alternative form of the *while* iteration-header is provided to improve readability by making it possible to attach loop-associated bookkeeping instructions directly to the header, rather than requiring these instructions to be placed remotely.

The instruction

(7) quit;

occurring either within a set-theoretical iteration or within a while-iteration is equivalent to

(8) go to L;

where L is a unique generated label occurring at a position immediately outside the scope of the iteration. We also allow this statement in several more explicit forms. The form

quit ∀x;

is equivalent to (8) where L is a unique generated label occurring at a position immediately outside the scope of the set-theoretical iteration whose header begins either with

(∀x ϵ ...)

or with

(min ≤ ∀x ≤ min, ...)

or with some other allowed form of range restriction in which "∀x" appears. Note that this generalized *quit* statement may cause control to be transferred out of several nested iteration scopes all at once. For example,

```
(∀x ϵ a) y = x; (while #y gt 2) (∀z ϵ y) n = n with z;
    if #n gt 10 then quit ∀x;; end ∀z; end while; end ∀x;
```

equivalent to

```
(∀x ϵ a) y = x; (while #y gt 2) (∀z ϵ y) n = n with z;
    if #n gt 10 then go to L;; end ∀z;;; <L:> ...
```

We also permit the forms

quit while;

and

(9) quit while t_1 ... t_n; etc.,

where $t_1,\ldots,t_n$ are the tokens which follow the keyword *while* in the header of the iteration from which exit is to be made.

The first of the above statements is equivalent to a transfer to a label occurring at a position immediately outside the scope of the innermost while-iteration containing the quit statement. The second, more explicit, *quit* statement has a significance which may be defined as follows. Let W be the innermost while-iterating header whose scope contains the *quit* statement and which begins with the sequence

(while t_1 t_2 ... t_n)

of symbols. Let L be a unique label occurring at a position immediately outside the scope of W. Then (9) is equivalent to the explicit go-to statement the explicit go-to statement *go to L*.

The instruction

(10) continue;

is used in a very similar way. If this instruction occurs either within a set-theoretical or a simple while iteration, it is equivalent to the go-to statement (8), where L is a unique generated label occurring within, but at the very end of, the scope of the iteration. Thus, for example,

```
(while x gt 0) x = x - f(x); if g(x) lt 0 then continue;
      else y = y + g(x);;  end while;
```

is equivalent to

```
(while x gt 0) x = x - f(x);  if g(x) lt 0 then go to L;
      else y = y + g(x);;  <L:>  end while;
```

Suppose next that a simple *continue* statement (10) occurs within a *while*-iteration whose header is of the more complex form

(while C doing blocka) .

Then, by definition, (10) causes a transfer to a unique generated label located immediately before the various statements of the block *blocka*, which (cf. (6)) form a group terminating the scope of the while-iteration. For example,

```
(while x ϵ s doing x = f(x);) if g(x) lt 0 then continue;
     else y = y + g(x);;  end while;
```

is equivalent to

```
(while x ϵ s) if g(x) lt 0 then go to L;
     else y = y + g(x);;  <L:> x = f(x);  end while;
```

We also allow more general *continue* statements having such forms as

(11) continue ∀x;

(12) continue while;

(13) continue while token;

This is how we define the significance of the statement (11). Let W be the innermost iteration header whose scope contains the continue statement (11) and which begins either with the sequence

(∀x ϵ ...)

or with some such sequence as

(min ≤ ∀x ≤ max, ...) .

Let L be a unique label occurring within the scope of this iteration, but at the very end of this scope. Then (11) is equivalent to

go to L; .

The significance of the statements (12) and (13) may be defined in similar fashion, and we leave it to the reader to supply the necessary details. Observe, however, that if the while-iteration header to which (12) or (13) refers contains a *doing* block, then (12) or (13) will cause a transfer to a label located immediately before the first statement in this block; note (cf. (6)) that this block of statements occurs at the very end of the scope of the *while* iteration.

We now describe a type of *compound operator* provided in SETL and related to the set-theoretic iterations with which we have just been concerned. If *op* is any binary operator or function of two variables, or more generally any expression having such an operator or function as its value, then

(14) $[\underline{op}:\ x_1 \in e_1,\ x_2 \in e_2(x_1),\ldots,\ x_n \in e_n(x_1,\ldots,x_{n-1})$
$\quad | \ C(x_1,\ldots,x_n)]\ S(x_1,x_2,\ldots,x_n)$

denotes the value v which would result from the following iteration calculation

(15) $v = \Omega;\ times = 0;\ (\forall x_1 \in e_1, x_2 \in e_2(x_1),\ldots,x_n \in e_n(x_1,\ldots,x_{n-1})$
$\quad | C(x_1,\ldots,x_n))$ if times $\underline{eq}$ 0 then times=1;
$\quad v = S(x_1,\ldots,x_n);$ else $v = v\ \underline{op}\ S(x_1,x_2,\ldots,x_n);;$

The "operator" appearing with brackets in (14) is called a compound operator. The construction (14) is subject to the same conditions concerning free occurrences of variables, etc., as is the iteration (2). In particular, numerical range restrictions of any of the forms (4A), (4B), (4C) are allowed. The construction (14) will often be used when the operator $\underline{op}$ is commutative and associative, in which case the value of (14) is independent of the order in which the set-theoretic iteration (15) is carried out. Note, for example, that the availability of the construction (14) allows us to write the ordinary mathematical formula

$$\sum_{x_1 \in a_1} \max_{x_2 \in a_2} \left| \bigcup_{x_3 \in a_3} S(x_1,x_2,x_3) \right|$$

as

$$[+:\ x_1 \in a_1]\ [\underline{max}:\ x_2 \in a_2]\ \#[\underline{u}:\ x_3 \in a_3]\ S(x_1,x_2,x_3)\ .$$

A resolving convention is required if the implied scope of a compound operator is not to be ambiguous. One may ask, for example, if the expression

$$[+:\ x \in a_1]\ f(x) + b$$

is to have the reading

$$([+:\ x \in a_1]\ f(x)) + b$$

or the reading

$$[+:\ x \in a_1]\ (f(x) + b)\ .$$

We adopt the convention that a compound operator is to be treated as a monadic operator; as such, it will have minimal scope (except insofar as built-in operators producing boolean from nonboolean quantities may have higher precedence). Thus, the first of the two possible readings noted above is correct.

We also permit *while* iterators to be used within compound operators, in much the same way that set-theoretic iteration headers are used. Thus, if C is a boolean-valued expression and *block* a block of statements, we may use

(16) [op: while C doing *block*] expr

to denote the value v which would result from the following iterative calculation

(17) v = Ω; times = 0; (while C doing *block*) if times eq 0
then times = 1; v = if times eq 0 then expr
else v op expr; end while;

We also permit *while*-iterators and set-theoretic iterators to be intermixed in a compound operator, so that, for example,

(18) [op: $x_1 \in e_1$, while C doing *block*, $x_2 \in e_2(x_1) \mid D(x_1,x_2)$] expr

is a legitimate expression whose value the reader will readily deduce.

2. Iterators over Tuples, Character Strings, and Bit Strings.

One will frequently wish to extend an iteration over all the components of a tuple, character string, or bit string. For this purpose, the iteration form

(1) (1 ≤ ∀n ≤ #tuple) x = tuple(n); block end ∀;

could be used. Since iterations like (1) would be used frequently, however, we provide the abbreviated form

(2) (∀x(n) ∈ tuple) block end ∀;

Iterators of the form (2) may also be intermixed with groups of other iterators, used in compound operators, existential and universal quantifiers, set-formers, and with quit and continue statements; in general the conventions explained in the preceding

paragraph apply *mutis mutandis* to iterators of the form (2).

Note the following simple applications. To form the set of components of a tuple (with no undefined components)

(3) set = {x, x(n) ϵ tuple}; .

To find the position of the first blank character in a string, if any:

(4) pos = if ∃c(n) ϵ string|c eq ' ' then n else Ω;

3. If-statements, Flow Statements.

SETL provides an ALGOL-like conditional statement which may have the form

if $bool_1$ then $block_1$ else if $bool_2$ then $block_2$...else $block_n$;

or may have the slightly simpler form

if $bool_1$ then $block_1$ else if $bool_2$ then $block_2$ ...

... else if $bool_{n-1}$ then $block_{n-1}$;

Here $bool_1,\ldots,bool_n$ are required to be boolean expressions; each of $block_1,\ldots,block_n$ is an arbitrary sequence of valid SETL statements, which may include go to statements and additional if statements.

Each statement block forming part of an if-statement, with the exception of the last such statement block, is terminated by the occurrence of the next following keyword *else* or *then*. The last block is terminated by the occurrence of the semicolon explicitly shown as terminating the if-statement displayed above. Of course, since the final statement of this final block is itself terminated by a semicolon, the visible sign of an if-statement termination will be a double semicolon. For example, we might have

if x gt 0 then set = set with x; else n = n+1;;

This style is acceptable for short blocks, but when long blocks of statements are encompassed within if-statements scopes delimited in this way would become confusing. For this reason, we permit a variety of alternative conventions. These are:

1. The use of parentheses. Any block of statements may be closed in parentheses, and remains a block of statements. Thus, we might write

```
if x gt 0 then set = set with x; else (n = n+1; k = k*n;);
```

2. The use of either

```
end;     or     end if;
```

or more explicitly still

```
end if t1 t2 t3 ... ;
```

where t_1 t_2 t_3 is the string of tokens following the keyword *if* in the statement being terminated. Thus, for example, one may write either

```
if x gt 0 then set = set with x; else n=n+1; k=k*n; end if;
```

or

```
if x gt 0 then set = set with x; else n=n+1; k=k*n; end if x;
```

If-statements may be nested, that is, an if-statement may occur as part of a *block* within another if-statement.

The conventional form of if-statement which we have just described is adequate in many situations, and by allowing if-statements to be nested within one another we significantly enhance the expressive power of this statement form. However, when complex sets of interlocking conditions must be dealt with, the nested if-statement becomes inadequate. The difficulty lies in the fact that the if-statement intermixes the controlling conditions of a subcasing operation with the transformations to be performed in the various subcases; such intermixture violates the fundamental design principle of *grouping by logical relation*. For this reason, SETL provides, in addition to the ALGOL-like if statement form defined above, yet another linguistic form, designed for the description of complex sets of conditional actions, and having the interesting property of exploiting the two-dimensional nature of paper. This statement form is introduced by the keyword *flow*, and will therefore be called a "*flow*-statement".

A flow statement consists of a *header* and a *trailer*. The header consists of the keyword flow, followed by a series of *header elements*, and terminated by a semicolon.

In the simplest form of *flow*-statement, only *test nodes* and *exit-action nodes* will appear, and each test node will consist simply of a name followed by the sign "?", while an exit-action node will consist merely of a name followed by the sign ",". (The comma immediately preceding the semicolon which terminates a flow-header is by convention omitted.) The following is an example of the somewhat unfamiliar construction we have in mind:

```
(1) flow                    nodeterm?
                arelexalts?                 islockey?
          findalt,   maynext,   keypres?              arealts?
                                aresecalts? maynext,   findalt, maynext,
                          findalt,  maynext;
```

The semantic intent of such a header may be explained as follows. Each test node N names a boolean expression to be evaluated. (The evaluation of this expression may be preceded by the execution of a block of statements, cf. below.) If this expression has the value true, then the *left-hand descendant* of N, i.e., the node immediately below and to the left of N, is evaluated (or in the case of an action node, performed; see below). If this expression has the value false, then control passes to the right-hand descendant of N. An action node names either a block of statements to be performed, or names a test node occurring elsewhere in the *flow* header, or names a label external to the whole flow statement text. In the first case the designated block of statements is performed, and control passes, in normal fashion, either to the next statement following the *flow* statement, or, in the presence of explicit *go to* commands, to some other statement. In the second case, control "loops back" to a prior test node; in the third case, control passes out of the flow statement to some other labeled statement. Note that the two dimensional "display" form of a *flow* header like (1) serves to make vivid the flow of control within a sequence of tests; in particular, it is easy to read off the collection of tests, and their outcomes, which precede any particular test or action in the tree.

Having made these heuristic comments, we continue with a formal scription of the structure of a flow statement, going on to describe its second or *trailer* portion. This part of a flow statement, which follows immediately after the header portion of the statement, serves to define the details of the tests and actions mentioned in the header portion of the statement.

The trailer part of a flow statement, as well as the flow statement itself, is terminated by the token sequence

(2) end flow; .

A flow statement trailer consists of a sequence of *definitions*, each of which is prefixed by a label identical with one of the names in the flow statement header. A label identical with one of the names occurring in the header will be called an *active* label. Each definition extends from the active label with which it begins up to either the next following active label or to the end of the entire flow statement, whichever is nearer.

The definition corresponding to each action node is a block of ordinary SETL statements. When control passes to the action node, this block of statements is executed, and control passes on in the manner described above.

The definition corresponding to a test node is a block of ordinary statements, to which a Boolean expression terminated by a semicolon and prefixed by the sign "=" is prefixed, this whole context therefore appearing as

(3) = expn;

When the test node is invoked, all statements in the definition corresponding to the test node are executed, following which the terminal Boolean expression (3) is evaluated. If this yields the value true, then control passes to the left-hand descendant of the test node; the value false transfers control to the right-hand descendant of the test node.

Action nodes not defined in the trailer of a flow statement may be called *exit* nodes; for clarity we allow such a node to be preceded by the word "to" in the case of an exit node. Consider, for mple, the following flow statement.

```
                flow                    t1?
                          act1,             to s2;
                t1:= x lt 0;
                act1: x=x+y;            end flow;
        s1:         ...;
        s2:         ...;
```

When this flow statement is encountered, the expression x lt 0 will be evaluated. If it has the value true, then the statement x = x+y will be performed, and control will pass to the statement labeled s1. On the other hand, if x lt 0 has the value false, then control will pass at once to the statement labeled s2.

The special name *quit* may be used in an action node; if, within a flow statement, a node so named is executed, control will pass to the first statement following the flow statement terminator.

The labels occurring in the trailer portion of a flow statement are assumed to be known only within the statement. That is, one cannot enter the tree in a nonstandard way by executing a go-to statement. Moreover, a go-to statement occurring within a definition in the trailer part of a flow statement must either reference a label (necessarily inactive), which is either part of the same definition or which lies entirely outside the flow statement. This rule serves to exclude "hidden" control flow, that is, control flow between the nodes of a flow statement which is not shown explicitly in the header part of the flow statement.

We now describe certain syntactic variations allowed within flow statements which improve the flexibility of this type of statement.

(a) In addition to test nodes and exit-action nodes, we allow *intermediate-action* nodes. Such a node consists of a name followed by the sign "+". The node name must be identical with a label appearing in the trailer part of the same flow statement; the definition following this label must be a block of ordinary SETL statements. The node will have precisely one descendant, which may be either a test node or another action node. When, in executing a flow statement, control passes to an intermediate action node, the code block defining this node is executed, following which control passes to the descendant node of the intermediate action node. (Of course, any explicit *go to*

encountered may modify this code flow.) Consider, as an illustration the construction just explained, the following flow statement

```
flow                    t1?
                   actl +      t2?
                   act3,     actl,  act2;
actl: ...;  t1:= ...;  t2:=...;  act2: ...;  act3: ... ;
end flow;
```

If *t1* has the value true, we perform actl, and then perform *act3;* if *t1* is not true we test *t2* and then perform either *act1* or *act2*.

(b) Any action node in a flow header may be preceded by an iteration-header. If such a prefix is attached to an action node, it is meant that the code in the node definition is to be executed iteratively in the manner defined by the iteration header. An iteration prefix used in this way does not call for a terminating semicolon. When the iteration terminates, the next action to be performed (which may simply involve passage of control to a successor node) is determined in the usual manner. For example, we may write

```
flow                    setnonnull?
       (∀s ∈ items(cls)) doelem,  printnullcase;
doelem: ...;
printnullcase: ...;
end flow;
```

In this case, the code block labeled by *doelem* will be executed iteratively, once for each element of the set *item(cls)*, following which control will pass out of the flow statement shown above.

(c) A node in a flow statement header may be replaced by the text which would otherwise appear in the definition corresponding to the node. In this case, the text of the definition must appear within parentheses. For example, we may write

```
flow  (x gt 0) ?
(y = x+1;),  (y gt 0)?
           on,          (z = x+y;);
on:  subr(x,y);  end flow;
```

(d) Any code block contained in a definition in the trailer part of a flow statement A may contain further flow-statements B, C, etc.,

but the exit transfers of these embedded flow statements must, in accordance with the rule for *go to*-statements stated above, refer e labels which are either part of the labelled definition or which lie entirely outside the (largest) developing flow statement A. A greater flexibility is provided by an explicit subflow option. To use this option, we interpolate one or more header extensions between the header of a flow statement and the trailer which follows it. Each such header extension consists of an *extension label* corresponding to some node named either in the preceding header or in some preceding header extension, followed by the keyword subflow, followed by a list of distinct *exit designators* separated by commas and appearing within parentheses, followed by text describing a flow structure and having precisely the structure of the text which appears within an ordinary flow statement header.

An example of a flow statement header with extension is as follows:

```
flow                    howcompare?
                  seenbefore?                    skip,
              skip,       process;
howcompare:subflow(bigout,smallout)      firstdefined?
                                  seconddefined?      definefirst+
                          firstbigger? definesecond+  seconddefined,
                bigout,   smallout, firstbigger;
```

This example illustrates the following syntactic and semantic points. Each header extension represents some sequence of tests and actions whose details one chooses not to show in the main header or preceding header extension within which this test/action sequence is first named. The header extension H labeled with a name NN appearing in a preceding test (or multi-test cf. below) node N is said to give detail concerning N. When control passes to N and the test-action sequence labeled NN is performed, the consequence (for control) will either be that exit is made from the entire enveloping flow statement, or that control passes to the subflow node labeled with a particular exit designator. If H gives detail concerning the node N, then the k-th exit designator in the sequ of designators which follows the keyword subflow is understood

refer to the k-th descendant of N. That is, if control passes to N, and if after the series of tests and actions which ensues control leaves H through its k-th exit designator, then it is understood that the next node to be executed is the k-th descendant of N.

Note in this connection that we allow a header extension to contain any number of exit designators, and accordingly allow the test of any header or header extension to contain test nodes having a number of descendants larger than 2. Such test nodes are called *multi-test nodes*, and their presence is indicated by writing

name ? k ,

where k is the number of descendants possessed by the multi-test node *name*. This number must of course be equal to the number of exit designators in the header extension labeled with the name *name*. This type of construction is illustrated by the following example.

```
flow                    inwhatrange? 3
    toosmall?            biggerunity?                    toolarge?
skip, biggerunity;  multiply,  divide,        skip,          multiply;
inwhatrange: subflow(lessa,middle,greaterb)  isgreaterb?
                                          greaterb,  islessa?
                                                lessa,   middle;
```

When control reaches the above flow header, the condition *isgreaterb* is first evaluated. If this has the value true, then the condition *toolarge* is evaluated. On the other hand if *isgreaterb* has the value false, then the condition *islessa* is evaluated. If *islessa* has the value true, then *toosmall* is evaluated; in the contrary case, the condition *biggerunity* is evaluated, etc.

The reader is invited to transpose the flow header shown above into some collection of nested if statements expressing the same logic, and to compare the clarity of text which results to that of the text shown above.

4. Subroutine and Function Definitions. Initialization.

SETL provides various features intended for the definition of subroutines and functions. A subroutine definition has the form

define name (arg1, arg2, ..., argn); body end name;

Here *name, arg1, ..., argn* are all valid names, while *body* is any valid sequence of SETL statements. The final statement

end name;

in the above definition locates its end. A subroutine defined in this conventional way is called via a statement

name(exprl,...,exprn);

in which *expr1,...,exprn* are expressions defining the actual subroutine arguments to be used at the moment of call. Return from a called subroutine is accomplished using a *return* statement having the familiar form

return;

Subroutines are always recursive, as are functions (see below).

Note that when a subroutine A is called recursively only the subroutine arguments and the variables *owned* by the subroutine are stacked; variables *owned* by other subroutines, but known within A (cf. the discussion of name scoping and the own declaration which follows) are not stacked. Thus, for example, a subroutine may be used recursively to add elements to a single particular external set.

Functions rather than simple subroutines are defined by writing

definef name(arg1,arg2,...,argn); body end name;

Return statements occurring within the body of a function definition must have the slightly expanded form

return expr;

here *expr* is an expression which, evaluated immediately before a return from the function, defines the function value.

In addition to these conventional forms of function and subroutine definition, SETL allows one definition of functions a...

subroutines called in a manner more closely resembling the normal TL use of infix and prefix operators. The definition of a function to be called as an infix operator is written as follows:

definef arg1 name arg2; body end name;

Aside from the fact that it is written in a different syntactic form, and that the function name is underlined, the above infix function has exactly the same semantic significance as any two argument function name(arg1,arg2).

A programmer-defined prefix or monadic operator will have only a single argument; its definition will have the form

definef name arg; body end name; .

Infix and prefix operators, as well as functions written in ordinary parenthetical style, may freely be used as parts of expressions. Monadic operators always have a precedence superior to that of any infix operators, except that built-in comparison and test operators such as gt, $\in$, etc. have higher precedence.

Subroutines of 1 and 2 arguments may also be written in infix and prefix form respectively, having in this case definitions of the form

define arg1 name arg2; body end name;

and

define name arg; body end name;

respectively. Subroutines defined in one of these forms should be called in the corresponding form.

We allow subroutines and functions to be called in a variety of ways generalizing the conventional (1).

The call

$$f[a_1,\ldots,a_n]$$

will return the set

$$\{f(x_1,\ldots,x_n),\ x_1 \in a_1,\ x_2 \in a_2,\ \ldots,\ x_n \in a_n\} \ .$$

This is consistent with earlier usage in case f is a set of ordered n+1-tuples. Still more generally, a call of the form

$$f(a_1,\ldots,[a_j],a_{j+1},\ldots,[a_k,a_{k+1},\ldots,a_\ell],\ldots)$$

will return the value

$$\{f(a_1,\ldots,x_j,a_{j+1},\ldots,x_k,x_{k+1},\ldots,x_\ell,\ldots), x_j \in a_j, x_k \in a_k, x_{k+1} \in a_{k+1},\ldots\}$$

For monadic operators we can write in very similar fashion and with similar meaning

$$\underline{op}[a];$$

For binary infix operators we can write

$$[a]\ \underline{op}\ b, \qquad a\ \underline{op}\ [b], \qquad [a]\ \underline{op}\ [b], \qquad \text{etc.}$$

We allow similar forms for subroutine calls. If f is a subroutine, then the call

$$f[a_1,\ldots,a_n];$$

is equivalent to the iteration

$$(\forall x_1 \in a_1, x_2 \in a_2,\ldots,x_n \in a_n)\ f(x_1,\ldots,x_n);;$$

the call

$$f(a_1,\ldots,[a_j],a_{j+1},\ldots,[a_k,a_{k+1},\ldots,a_\ell]\ldots);$$

is equivalent to the iteration

$$(\forall x_j \in a_j,\ x_k \in a_k,\ x_{k+1} \in a_{k+1},\ldots)$$

$$f(a_1,\ldots,x_j,a_{j+1},\ldots,x_k,x_{k+1},\ldots,x_\ell,\ldots);;$$

Subroutines and functions are legitimate atom-types within SETL; in particular, they may validly be elements of sets, components of ordered pairs, arguments of other subroutines, etc. This fact allows various powerful programming devices to be used; we may, for example, tag a set with certain functions of access and combination which are associated with it, etc.

Aside from the operation of application that naturally belongs to subroutines and functions, the only built-in operations which apply to atoms of these types are the boolean tests eq and ne of equality and inequality.

We regard a subroutine definition as *initializing* a variable with name identical to the subroutine name. The general conventions applying to initializations are as follows.

Each statement in a block of statements occurring in the form

(1) initially block;

will be executed the first time the subroutine containing (1) is ıtered, but not subsequently. If any such statements occur within a subroutine (or function), they must precede all other statements of the subroutine (with the exception of declarations associated with name-scoping; see below). This gives us a general method for the initialization of variables.

A few extra words defining the semantics of initialization blocks (1) somewhat more precisely are in order; precise conventions are especially needed when (1) occurs within a recursive subroutine. We take the initialization (1) to be precisely equivalent to the statement

(2) if flag eq 0 then flag = 1; block;

where *flag* is a generated variable uniquely associated with the initialization statement (1). The variable *flag* occurring in (2) is taken to be external to the subroutine A in which (1) occurs i.e., this variable is not stacked if A is entered recurisvely.

Thus the statements of *block* will be executed only when A is invoked for the first time, even if A is entered recursively. On the other hand, perfectly arbitrary statements, and even recursive subroutine calls, may be contained in *block*.

In addition to the effects of *explicit initializations* of the form (1), the values of variables will be initialized in the usage-defined situation described below; such *implicit initializations* will be made before any *explicit initializations* of the form (1) are made. Implicit initialization is performed:

i. if a name *labname* is used as a label within a SETL routine, then the variable of the same name will be initialized to have a value equal to the unique label atom corresponding to the label *labname*. This initialization is useful in that it allows us to write such expressions as

```
go to {<const1,lab1>,<const2,lab2>,<const3,lab3>}(expr);
```

and thus to make use of "calculated go-to's" in flexible and convenient form.

ii. certain very important initializations are connected with ɜ use of names as procedure names (i.e., as the names of subroutines or functions). These will be explained in more detail below.

However, before taking up these points, we must explain the general SETL namescoping rules which form their background.

5. Additional examples of the use of SETL.

Various important features of the SETL language, notably its namescoping rules, still remain to be explained. However, since specifications of language features are dry and ultimately confusing if the actual use of the language is not illustrated by examples, we pause in our systematic account of SETL to give a few examples.

A. Elementary examples:

We begin by giving a few elementary definitions of functions to be used later, thereby illustrating, among other things, the definitional facilities of SETL. The functions defined in the following section will be used in later discussion.

The following insertion and selection-removal operators are useful.

```
define a in b;  b = b with a;  return;  end in;
define x from s;  x = ɔs;  s = s less x;  return;  end from;
```

This last subroutine chooses an arbitrary element from a set, removing the element from the set at the same time.

The following useful function, which we write in infix form, assigns a specified value to a quantity if the quantity happens to be undefined:

```
definef val orm val2; return if val ne Ω then val else val2;
                                    end orm;
```

The very general sinister call conventions described earlier in the present section can be used to define various functions useful in the presentation of pushdown stacks as tuples:

```
definef top tup; return tup(#tup); end top;
```

```
definef newtop tup;
  (load) x = tup(#tup);  tup=tup(1: #tup-1); return x;;
  (store x) tup(#tup+1)=x; return;;
end newtop;
```

The following function reverses a tuple:

```
definef rev(tupl); return[+: #tuple≥j≥1]<tupl(j)>;  end rev;
```

If a function f is defined by a set of ordered pairs, the simple expression rev[f] gives the inverse function.

B. Sorting.

Occasionally, given a set s and a numerical function f defined on s, one wishes to sort the elements of s according to increasing values of f. The following procedure assigns an element of s its position place(s) in sorted order.

```
place = nl;
(∀x ∈ s)
   place(x) = #{y ∈ s|f(y) lt f(x) or
      (f(y) eq f(x) a place(y) ne Ω)} + 1;
end ∀;
```

More plausible sorting algorithms may also be represented in SETL. Here is the slightly better *insertion sort*, which sorts a sequence of n elements, in place, in a number of steps proportional to n^2.

```
(1 < ∀j ≤ #seq)
   (j > ∀k ≥ 1)
      if seq(k+1) lt seq(k) then
         <seq(k),seq(k+1)> = <seq(k+1),seq(k)>;
      else
         quit ∀k;
      end if;
   end ∀k;
end ∀j;
```

The following algorithm defines the *bubble sort*, which is about as efficient as the sorting method just described.

```
n = 1;
flow                          attop?
                        'quit,      reversed?
                                inter+   (n=n+1;)+
                                atbot?    attop,
                         (n=n+1;)+   (n=n-1;)+
                          attop,      attop;
attop := n eq #seq;
reversed := seq(n+1) lt seq(n);
inter: <seq(n+1),seq(n)> = <seq(n),seq(n+1)>;
atbot := n eq 1;
                end flow;
```

The simple insertion sort algorithm described above will operate most efficiently if presented with data actually in order. The *sliding insertion sort* or *shell sort* exploits this fact, incorporating a device which causes an array being sorted to converge rapidly to approximate order. The algorithm is as follows. A descending sequence $M_n, M_{n-1}, \ldots, M_1$ of integers is chosen, with $M_1 = 1$. Successively, for each j from n to 1, the array $\{a_k\}$ to be sorted is divided into M_j subsequences

$$a_1,\ a_{M_j+1},\ a_{2M_j+1},$$

$$a_2,\ a_{M_j+2},\ a_{2M_j+2},\ \cdots .$$

Each of these arrays is sorted using the ordinary insertion sort algorithm. Note that when j = 1 we have $M_1 = 1$, so that the ordinary insertion sort algorithm is eventually applied to the whole array, guaranteeing that complete order is eventually obtained. Experience shows that it is advantageous to define the sequence $M_n, M_{n-1}, \ldots, M_1$ as follows: put $M_n = 2^k - 1$, where 2^k is the largest power of 2 not exceeding the number of elements to be sorted, and then put $M_{j-1} = M_j/2$. In SETL, all this may be written as follows:

```
m=1;  (while m le #seq) m = 2*m;; m = m-1;
(while m gt 0 doing m = m/2;)
   (1 ≤ ∀j ≤ #seq) k = j-m;
      (while k gt 0 and seq(k+m) lt seq(k) doing k = k-m;)
         <seq(k),seq(k+m)>=<seq(k+m),seq(k)>;;
end ∀j;  end while m;
```

Next we describe the so-called *tree insertion sort*. In this method a binary tree, to whose nodes the elements to be sorted are attached, is built up by attaching successive branches. The tree is built in such a way as to ensure that if an element x is attached to a particular node N, then x exceeds all the elements attached to the left-hand sub-tree of N, and is exceeded by all the elements of the right-hand sub-tree of N. The rule for attachment of a new element x is as follows. Examine successive nodes y, beginning at the tree root. If x exceeds y, move down the tree to the right; or if y has no right descendant, make x the right descendant of y. If y exceeds x, move down the tree to the left, or if y has no left descendant make x the left descendant of y. When all the elements of the array to be sorted have been attached to the nodes, 'linearize' the tree to an array by the following recursive rule: first linearize the left-hand sub-tree; then take the element attached to the tree root; then linearize the right-hand sub-tree to get the top part of the sorted array. This procedure will on the average sort an array of n elements in a time proportional to n log n. In SETL, it appears as follows:

```
r = nl;  l = nl;  elt = nl;  ntop = newat;  elt(ntop) = seq(1);
(1 < ∀j ≤ #seq) x = seq(j);  top = ntop;
      flow                    xbigr?
                    isright?          isleft?
               down+  hangright,   down+  hangleft,
               xbigr,              xbigr;
      xbigr  :=  x gt elt(top);
      isright  := (r(top) is desc) ne Ω;
      isleft  :=  (l(top) is desc) ne Ω;
      down:  top = desc;
      hangright:  r(top) = newat;  elt(r(top)) = x;
      hangleft:   l(top) = newat;  elt(l(top)) = x;  end flow;
end ∀j;
seq = nl;  traverse ntop;
define traverse top;
/* seq, elt, l, and r are all global and owned by a procedure
   external to this routine */
```

[continued]

```
if top eq Ω then return;;
traverse ℓ(top);  seq(#seq+1) = elt(top);  traverse r(top);
return;  end traverse;
```

The *simple selection* sort is as follows: survey the n elements of a set to be sorted to find the minimum element; remove it from the array; and iterate. In SETL:

```
sorted=nℓ;  (while set ne nℓ)
sorted(#sorted+1)=[min:x∈set] x is minelt; set=set less minelt;
end while;
```

A variant of this basic idea yields the much faster *tree selection sort*, which may be described as follows. First, attach the elements of the array to be sorted as the twigs of a binary tree of appropriate size. Next, propagate values up to the tree root, attaching to each node the minimum of the values attached to its immediate descendants, and causing each node to point to that immediate descendant node to which this minimum value is attached. When this structure is built, it becomes trivial to locate the original array minimum, detach it from the tree, and move it to a workspace in which sorted array elements are accumulated. After this removal operation, the tree is repaired by redetermining a minimum-of-descendants value for all nodes above the node just removed; after which the selection, removal, and repair process iterates until completion.

Adopting the convention that the minimum-of-descendants is always found down the left-hand branch, the following shows a way that the above algorithm may be written in SETL.

```
/* first build the tree */
ℓ = nℓ; r = nℓ; v = nℓ; par = nℓ; trees = nℓ;
(1 ≤ ∀j ≤ #seq) newat is node in trees; v(node) = seq(j);;
(while (#trees) gt 1 doing trees = newtrees;) newtrees = nℓ;
   (while (#trees) gt 1) ln from trees; rn from trees;
      newat is nd in newtrees; par(ln) = nd;  par(rn) = nd;
      if v(ln) gt v(rn) then <ln,rn> = <rn,ln>;;
      <ℓ(nd), r(nd), v(nd)> = <ln, rn, v(ln)>;
   end while;
   if trees ne nℓ then ∃trees in newtrees;
end while;
```

```
/* now tree is built; begin main selection and repair process */
top = ∋newtrees;  seq = nℓ;
 while ℓ(top) ne Ω) node = top;
   (while ℓ(node) ne Ω) node = ℓ(node);;
   seq(#seq+1) = v(node);  ℓ(par(node)) = Ω;
   (while par(node) ne Ω) node = par(node);
      flow                    isldesc?
                 isrdesc?                 maker +
          isℓbigr?  fixv,                 fixv,
     switch+  fixv;
     fixv;
isℓdesc :=  ℓ(node) is ℓdesc ne Ω;
isrdesc :=  r(node) is rdesc ne Ω;
isℓbigr :=  v(ℓdesc) gt v(rdesc);
switch  :  <ℓ(node), r(node)>  =  <r(node),ℓ(node)>;
maker   :  <ℓ(node), r(node)>  =  <r(node),Ω>;
fixv    :   v(node) = v(ℓ(node));                          end flow;
   end while par;
end while ℓ(top);
```

The still more remarkable *heapsort* remedies certain of the deficiencies of the tree selection sort, and provides a method for sorting an array in place and in a number of steps bounded by n log n. It is in essence a binary tree selection sort in which the tree pointers are implicit, the descendants of the element at array location j being the elements at locations 2j and 2j+1. The algorithm is as follows.

```
(1 < ∀n ≤ #seq) m = n;
   (while if m le 1 then f else seq(m/2) lt seq(m))
     <seq(m), seq(m/2),m> = <seq(m/2),seq(m),m/2>;
   end while;  end ∀n;
(#seq ≥ ∀top > 1) <seq(1),seq(top)> = <seq(top),seq(1)>; m = 1;
   (while 2 * m lt top doing m = targ;)
      targ = if seq(2*m) lt seq(2*m+1) and 2*m+1 lt top
         then 2*m+1 else 2*m;
      if seq(m) ℓt seq(targ) then
         <seq(m),seq(targ)>=<seq(targ),seq(m)>;
      else quit;
      end if;
   end while;
end  ∀top;
```

Quicksort is a high-speed descendant of the simple bubble sort It operates as follows: take the first element x of an array a and, as in the bubble sort, compare it to its successor y, interchanging x and y whenever x exceeds y. However, if y exceeds x, interchange y with the element z having the largest possible index consistent with the assumption that x exceeds z. As this process proceeds, an increasing region R_- of elements known to be less than x will build up above x, and an increasing region of elements R_+ known to exceed x will build up below x. Eventually x will come into its proper place. Then, if either R_- or R_+ contains just two elements, they may be placed in order by a single interchange; in the contrary case, the procedure just described may be used recursively to sort R_- and R_+.

In SETL, quicksort appears as follows:

```
define quicksort(a,i,j); /* sorts part of array a between a(i)
   and a(j) */
flow                (i ge j)?
        (return;),                  (i eq (j-1))?
                        interchange2+          (lowinx = i;hyinx=j;)+
                        (return;),             lowinxlesshy?
                                    nextsmaller?              sortparts+
                         interchange+      pushup+            (return;),
                         lowinxlesshy,     lowinxlesshy;
interchange2 : if a(i) gt a(j) then <a(i),a(j)>=<a(j),a(i)>;;
lowinxlesshy := lowinx lt hyinx;
nextsmaller  := a(lowinx) gt a(lowinx+1);
interchange  : <a(lowinx), a(lowinx+1),lowinx>
                    = <a(lowinx+1),a(lowinx),lowinx+1>;
pushup       : <a(hyinx),a(lowinx+1),hyinx>
                    = <a(lowinx+1),a(hyinx),hyinx-1>;
sortparts    :  quicksort(a,i,lowinx-1 );  quicksort(a,lowinx+1,j);
                                                    end flow;
end quicksort;
```

Merging procedures of various kinds play a central role in many of the most important methods for sorting large arrays by using es Fast internal sorts can also be built using merge techniques. We shall describe one such sort, the so-called *natural two-way merge*.

It works as follows: given an array of elements to be sorted, use a workspace of equal size, and merge elements from the top and bottom the array into the bottom of the workspace as long as these elements may be used to form an increasing sequence or *run*. Naturally, if both the top and the bottom element can be used to continue a run, we first use whichever is smaller. When a run cannot be continued, we start a new run, placing it in reverse sequence of positions at the top of the workspace, until once more the run can no longer be continued. At this point, start a new run, storing it at the bottom of the remaining workspace area, etc. When the whole of the original array has in this way been transformed to the workspace, interchange the array and workspace, and repeat. During this process, the number of separate runs into which the total array is divided will be cut in half each time the whole array is combed through, and eventually complete order will result.

In SETL, this procedure appears as follows.

```
start:  bot = 1; top=#elt; xbot=1; xp=top; flag=-1; extra=nult;
onward: if flag eq -1 then  xtop=xp; xp=xbot;
        else xbot=xp; xp=xtop; end if;
        flag = -flag;
        if elt(bot) le elt(top) then
           extra(xp) = elt(bot);  bot = bot+1;
        else
           extra(xp) = elt(top); top = top-1;
        end if;
        (while top ge bot doing xp=xp+flag;)
           flow            topbest?
                usetop,              botok?
                                usebot,  endrun;
topbest := elt(top) ge extra(xp) and elt(top) le elt(bot);
botok   := elt(bot) ge extra(xp);
usetop  :  extra(xp+flag) = elt(top); top=top-1;
usebot  :  extra(xp+flag) = elt(bot); bot=bot+1;
           end flow;
        end while;
endrun  :  if top ge bot then go to onward;;
           elt = extra; if xtop lt #elt then go to start;;
```

Distribution or *pocket sorts* are, in a certain sense, dual to merging sorts. The *simple pocket sort* is the method used to sort punched cards on electromechanical equipment. The algorithm is

as follows. A given collection of keys is to be sorted. One regards these keys as integers to some base p, i.e. as sequences $d_1 \ldots d_k$ of base p digits. The items to be sorted are then distributed into p piles or pockets, according to the least significant digit d_k. Then the piles are gathered up into a single sequence, being taken in the order 0,1,...,p-1, and the distribution process repeated, first for the digit d_{k-1}, then, after regathering, for the digit d_{k-2}, etc. The relative positions assigned in this method to two items during the j-th distribution pass will not subsequently change except as required upon examination of more significant key digits during a later pass; and thus fully sorted order must emerge when all the digits of the keys have been processed.

We may represent this algorithm in SETL as follows.

```
multi = t;  q=1;  pocket=nult;
(while multi doing q = q * p;)
   seq=[+:x(n) ∈ dist(seq,p,q,multi)]x;
end while;
definef dist(seq,p,q,multi); multi = f;
(0 ≤ ∀n < p) pocket(n+1) = nult;;
(∀s(k) ∈ seq) key = ((s/q) is keyhead) //p+1;
   pocket(key) = pocket(key) +  s ;
   if keyhead ge p then multi = t;;
end ∀;
return pocket;  end dist;
```

The *radix exchange sort* is another fast key sort. It works as follows. Regard each item of the array to be sorted as a boolean string. On a first pass through the array, and by performing appropriate exchanges, place all elements whose lead bit is zero at the bottom of the array, and all elements whose lead bit is one to the top of the array. Then, recursively, apply the same procedure to the top and bottom of the array and to the second through last bit of each item.

In SETL, letting *b* be the number of bits in a key, this procedure appears as follows (note that in this example test conditions are directly embedded in the flow tree; cf. section 3 above):

```
/* to sort seq , use the following call: */
radsort(seq,1,#seq,b);
define radsort(seq,bot,top,bp);if bot ge top then return;;
   i = bot; j = top;
```

```
(while i lt j) flow     (seq(i)(bp))?
                    (seq(j)(bp) )?  (i=i+1;),
                (j=j-1;),   interchange;
interchange: <seq(i),seq(j)> = <seq(j),seq(i)>;<i,j>=<i+1,j-1>;
                                        end flow;
end while;  mid=if seq(i)(bp)then i-1 else i;
radsort(seq,bot,mid,bp-1); radsort(seq,mid+1,top,bp-1); return;
end radsort;
```

The *Ford-Johnson Tournament sort* reduces, to a level very close to a known lower level, the number of comparisons required to sort n elements. However, the number of moves required will on the average be proportional to n^2. This method is therefore of interest only in the unlikely but conceivable special case in which the cost of comparing items is so high relative to the cost of moving them that attention really does focus exclusively on comparisons. The algorithm is as follows. Divide the items to be sorted into n/2 pairs, arranging each pair so that its first element exceeds its second. Then sort the pairs according to their first elements, using the tournament sort procedure (recursively). This produces a pair of arrays $a_1,\ldots,a_m$ and $b_1,\ldots,b_m$, where $m = n/2$, where the a's are in increasing order, and where $a_j \geq b_j$ for all j. Finding the proper position of one new element among p already ordered others involves q comparisons, where q is the smallest integer such that $2^q-1 \geq p$; and such a location process is at its most efficient when p is precisely of the form 2^q-1. So then

b_1,a_1,a_2 form an ordered sequence of 3 elements, into which b_3 may be inserted using two comparisons;

The set of elements b_1, a_1 and b_3 in proper position are then 3 in number, and b_2 may be inserted among them using two comparisons;

The set of b_1,a_1,a_2,a_3,a_4, together with b_2 and b_3 in proper position, are then seven in number, so that first b_5, and then b_4 may be inserted into position using 3 comparisons.

$,\ldots,b_5$, together with $a_1\ldots a_{10}$, are then 15 elements, and thus b_{11}, and then $b_{10},\ldots b_6$ may be inserted into position using 4 comparisons,

and so forth.

This rather complex sorting algorithm may be written in SETL as follows; we assume a sequence *items* is given for sorting, on whic a user-supplied function le(x,y) is defined, which is "true" if x≤y.

```
definef fordj(items);
if(#items) eq 1 then return items;;
au = nl; bu = nl; i = 1; /* unsorted a and b sequences.*/
(while i lt #items doing i = i+2;)
    x = items(i);  y = items(i+1);
    if le(x,y) then <x,y> = <y,x>;;
    au(#au+1)=x; bu(#bu+1)=y; end while;
oddone = items(i); /* only exists if #items is odd. */
a = fordj(au);     /* sort the half-length sequence.*/
(1 ≤ ∀j ≤ #a       /* rearrange bu in the same way that au was  */
    dummy = 1 ≤ ∃n ≤ #a|au(n) eq a(j);        /* rearranged.   */
    au(n) = Ω;
    b(n) = bu(j);  end ∀j;
b(#b+1) = oddone;
/* now merge the components of 'b' into 'a' using a binary search.
    sequence 'a' will grow on the left, with its lowest index (lia)
    becoming negative */
lia=1; jbot=1; jtop=1; length=1;
(while jbot le #b)
    (jtop ≥ ∀j ≥ jbot)
        /* merge b(j) into the sequence a(lia:j-1) */
        low = lia-1;  high = j;
        (while(high-low) gt 1)
            mid = high+low /2;
            if le(b(j),a(mid)) then high=mid; else low=mid;;
        end  while;
        /* b(j) goes between low and high (even in the cases
        where it goes on an end, as in the case of b(1).  */
        (lia ≤ ∀i ≤ low) a(i-1) = a(i);;
        a(low) = b(j);  lia = lia-1;  end ∀j;
    jbot = jtop+1;  length = 2*length+1;  jtop=(lia+length)min #b;
end while;
return {<p(1) - lia+1, p(2)>, p ∈ a};  /* make it 1-origin */
end fordj;
```

6. Namescoping, Variation of references caused by recursive subroutine calls and returns. Initialization rules applying to subroutine names.

SETL provides a family of namescoping mechanisms, which it is hoped are sufficiently general and powerful to be convenient in the development of very large systems of programs. Of course, only experience not presently available can testify to the success (or failure) of the scheme proposed. It is hoped also that the proposed conventions will support user languages with a useful variety of user-level namescoping conventions. Here also, more experience is required.

We regard a namescoping system as a set of conventions which assign a unique 'resolved name' y to each 'source name' x appearing in a mass of text. The particular y to be assigned to each occurrence of x depends on the location of y within a nested, tree-like family of *scopes*.

The purpose of a namescoping system is of course to balance the conflicting pressures toward globality and protection of names. Unrestrictedly global use of names is unacceptable, since it creates a situation of 'name crowding' in which names once used become, in effect, reserved words for other program sections. Hard-to-diagnose 'name overlap' bugs tend to abound in such situations. 'Globalization' of any subcategory of names can recreate this problem. For example, in large families of subroutines it may become difficult to avoid conflicts between subroutine names. In sufficiently large program packages, it will be desirable to give even major scope names a degree of protection.

On the other hand, a system in which names tend very strongly to be local unless explicitly declared global can tend to force one to incorporate large amounts of repetitive declaratory boilerplate into almost every protected bottom level namescope or subroutine. In a language like SETL, which aims at the compressed and natural statement of algorithms, this burden is particularly irritating.

What we therefore require is a system capable of dividing a potentially very large collection of programs into a rationally anized system of 'sublibraries', between which coherent cross-referencing is possible in a manner not requiring clumsy or elaborate locutions.

More specifically, a namescoping scheme for SETL must address the following problems:

i. All function and subroutine calls in SETL are recursive. If a routine is called before returns from all previous invocations have been executed, we must know which variables should be stacked prior to entry. These are the variables which are said to be owned by the routine. Conventions unambiguously determining ownership of variables are required. These conventions must apply not only to names used as ordinary variables within procedures, but also to names used as procedure arguments, store-block arguments, and as labels. Similar issues arise in connection with names used as macros.

ii. A procedure p_1 will occasionally wish to access variables owned by a second procedure p_2. Our namescoping system will therefore have to include rules specifying when an occurrence of a name x in p_1 references the same quantity as an occurrence of x in p_2. Moreover, p_1 will occasionally wish to reference the x of p_2 using a local name y distinct from x. For example, this will be necessary if distinct variables, both initially called x, but occurring in two distinct procedures p_2 and p_3, are to be accessed from within p_1. Thus our namescoping scheme will not only allow inter-procedure references, but will also support some degree of 'name aliasing' in connection with these references.

iii. Procedure names will normally be used in a more global manner than names designating variables used within procedures. Rules defining the situations in which distinct names reference a single procedure, or in which identical names occurring in different scopes reference distinct procedures, are required. Similar remarks may be made concerning scope names themselves.

iv. Since in SETL variables may have procedures as values, and since a single variable may at different moments have different procedures as its value, we regard each procedure definition as a type of initialization. Our namescoping scheme must define the initial value of each variable occurring as a procedure name.

v. In the absence of appropriate restrictions, the degree of freedom in referencing implied by i-iv above would make all

too easy the introduction difficult-to-find bugs connected with mote references. To prevent this, our namescoping scheme must ınvolve restrictions which make unlikely the inadvertent use of patterns of remote reference which substantially change the meaning of a given section of code. That meaning which a code passage seems to have should be the meaning which it does have, even in the presence of remote references. We shall call any violation of a restriction forming part of the SETL namescoping scheme a scoping error. Various types of scoping error will be pointed out as the details of restrictions are given.

It is hoped that the namescoping scheme which will now be presented addresses these complex issues adequately. At any rate, since in the present section quite a number of basic semantic matters must be treated all at once, careful exposition, and patient attention on the reader's part, are both in order.

Certain important characteristics of the SETL name resolution conventions are noted in the following remarks, intended as introduction to the detailed namescoping specifications given below.

a. We deliberately break the conventionally very close connection between subroutine boundaries and name scopes. Thus name scopes enclosing several subroutines are allowed; at the same time, a single subroutine may contain several independent name scopes.

b. We regard scope boundaries as logical 'brackets' possessing a certain power to protect names within them from identification with names of the same spelling located outside. For flexibility, distinct numbered levels of bracketing are provided. We stipulate that, within a scope, two variables with different names are different unless an explicit declaration is made.

c. We provide mechanisms for identifying variables which appear in the same scope and have different names, or appear in different scopes. The mechanisms for identification act recursively. Two methods are provided for the identification of variables appearing in different scopes. Variables can be identified either by being made *global* within a scope s, in which case they are transmitted to

scopes included within s, or by using explicit remote references (see the *include* declaration discussed below).

d. An item *i* occurring in a namescope *ns* can only be identified with an item *j* occurring in a different namescope *ms* if *i* becomes 'known' in *ms* (or, conversely, if *j* becomes known in *ns*). In the namescoping scheme to be presented, it is actually this notion, that of an item occurring in one namescope becoming known in another, that is fundamental; given this notion, the rules for identification may be defined in a somewhat corollary way.

e. Each SETL subroutine or function is taken *ipso facto* to be a namescope, of level 0. Note that such a scope, like any other scope, can both contain embedded scopes and be contained in a larger scope.

We now begin to present the SETL namescoping scheme in detail.

The text of a SETL program consists of a linear sequence of tokens, grouped into a nested family of namescopes (which for brevity we may refer to simply as scopes). A scope is opened by a header line having the form

(1) scope <(optional) level indicator> <scopename>;

for example

scope 3 optimizer;

Here, <scopename> designates a simple or compound name, which names the scope. The optional <level indicator>, if it occurs, has simply the form

<integer> or - <integer> .

The nonoccurrence of a level indicator is logically equivalent to the occurrence of a level indicator with a value of 1. A scope opened by the header line (1) is closed by the occurrence of a matching trailer line

(2) end <scopename>;

for example

end optimizer;

All the text included between (1) and the next following matching line (2) constitutes the body of the scope headed by (1). A lir

(2) matching each line (1) is required; the absence of a matching trailer constitutes a scoping error. Several other forms of scoping error will be described in the following paragraphs; a text is acceptable to the namescope processor only if it contains no scoping errors.

The text comprising a scope *ns* falls naturally into several portions:

(a) imbedded subscopes;

(b) scope-associated declaratory text (to be described in more detail shortly);

(c) other text, which we call the proper text of the namescope *ns*.

This proper text is of course SETL code defining various SETL subroutines, functions, etc.

The beginning of a scope *ms* imbedded within another namescope *ns* is marked by the occurrence of a header line of the form (1); if such a header line occurs in *ns*, we require that a matching trailer line (2) be present in the body of *ns* (condition of well-formed nesting). In such a case, we call *ms* a subscope of *ns*. We say that *ms* is directly imbedded within *ns* if *ms* is a subscope of *ns*, but is not a subscope of any (proper) subscope of *ns*. In this case we call *ns* the parent scope of *ms*, and call *ms* an immediate descendant of *ns*. If several scopes have the same parent scope, they are said to be siblings of one another.

We require that scopes have names differing from the names of their parents and the names of each of their siblings. This allows us to refer to each scope in a unique manner by using a sufficiently long name string formed by concatenating the scope's immediate name with the name of its parent, its parent's parent, and so forth. Thus, for example, in a sufficiently large program library the following configuration of scopes might occur:

(3)
```
scope linearprogramming;
      scope optimizer;
      x = ...
      end optimizer;
      ...
end linearprogramming;
```

[continued]

(3) [continued]

```
scope fortrancompiler;
    scope optimizer;
    x = ...
    end optimizer;

    ...
end fortrancompiler;
```

In the discussion which follows we shall, in order to refer unambiguously to one of the two different scopes called *optimizer* use "hyperqualified" names of the form 'optimizer.fortrancompiler' and 'optimizer. linearprogramming'. Note, in connection with the above example, that this allows us to refer unambiguously to two different scopes, both called 'optimizer'.

Similarly, two distinct variables named x, occurring within these distinct scopes, can be distinguished by using the hyperqualified names 'x.optimizer.fortrancompiler' and 'x. optimizer.linearprogramming' .

Within the total mass of proper text (cf. (c) above) associated with a namescope *ns*, various tokens will occur. For the purposes of the following discussion, it will be convenient to designate each such occurrence of a token t by a symbol showing explicitly the nest of scopes in which t appears. For definiteness, we will write this symbol as

(4) $$t.\ ns_1.\ ns_2.\ ns_3.\ \ldots\ .ns_k$$

where $ns_1,\ldots,ns_k$ is the nest of scopes containing t, ns_1 being the smallest such scope; ns_2, the parent of ns_1; ns_3, the parent of ns_2; etc. The final scope ns_k is an 'outermost' scope, and hence a scope possessing no parent. A symbol (4) will be called an item.

The item propagation rules to be described in the following paragraphs will make items occurring in one scope known in other scopes. (This is the immediate effect of the include and global declarations to be described shortly.) Any item known in a namescope *ns* is known there under some local alias. (The rules determining the local aliases within *ns* of items known there but not occurring there will be explained below.) Identical items (always reference the same object; beyond this, the central problem

of any namescoping scheme is to decide when two token occurrences, t designated by the same item symbol, reference the same object. The present namescoping scheme uses the following fundamental rule to make this decision: if within *ns* there occurs an item (4), and if an item i' not occurring within *ns* becomes known within *ns* under the local alias t', then (4) and i' designate the same object if either the t of (4) is an initial part of the compound token t', or vice versa.

In the rule just stated we meet the important notion of 'compound token' for the first time; of course, a definition of this notion is required immediately. A simple token is an item recognized as integral by the lexical scanner for SETL; this may be either a special symbol, constant, simple name, underlined name, etc. A compound token is a sequence of simple tokens connected by occurrences of the 'underbar' symbol. Thus

xl

is a simple token, while

xl_scope1_chapter3

is a qualified token. Similarly,

\+ and maxop

are simple tokens;

+_scope1_chapter3

and

maxop _scope1_chapter3

are compound tokens. The successive simple tokens making up a compound token are its parts. The lexical type of a compound token is the lexical type of its first part. With the possible exception of its first part, every part of a qualified token must be a simple name.

Both simple and compound tokens are allowed to designate variables, procedures, etc. in SETL programs. However, compound tokens play a specific role in connection with the SETL namescoping scheme. More precisely, the local alias under which an item becomes known in a name scope *ns* distinct from the scope *ms* in

which it occurs will always be a compound token. In the manner defined by the fundamental rule stated above, the structure of this compound token will then govern the identification of the item (4) with an item occurring in the namescope *ns*.

For example, under the rules to be explained below an item *i* may become known in a namescope *ns* under the local alias

(5) x_optimizer_linear .

Then a reference within *ns* having either the form

(5') x_optimizer

or

(5") x_optimizer_linear

or simply

(5"') x

will reference the same object as *i*.

It deserves to be mentioned that we use '.' to separate scope names (as in (4)) only in the present "meta-discussion" of namescopes. The SETL user will employ compound names involving underbars only. In the present meta-discussion, the use for different but related purposes of the two different punctuation marks '.' and '_' prevents ambiguities of reference that could otherwise arise. Suppose, for example, that we wished to discuss text containing the following lines:

(6)
```
scope programming;
      scope optimizer_linear;
      x = ...
      end optimizer_linear;
end programming;

scope linear_programming;
      scope optimizer;
      x = ...;
      end optimizer;
end linear_programming;
```

The first x in (6) is referenced in the present meta-discussion as

x.optimizer_linear.programming ,

and the second x in (6) as

(7') x.optimizer.linear_programming .

If two separate punctuation marks were not available, these references would be identical.

Two principal declaratory forms, a global declaration and an include declaration, are provided in the SETL namescoping scheme. The global declaration allows items to be propagated from one namescope *ns* to other namescopes physically included within *ns*. It thus achieves effects similar to those achieved by the namescoping schemes used in ALGOL-60 and PL/1; however, globality is less 'automatic' in the SETL scheme than in the scheme provided by either of these two languages. The include declaration allows items to be propagated very selectively between namescopes standing in no particular relationship of physical proximity. In this regard it resembles FORTRAN 'COMMON'; however, our include conventions are considerably more systematic and general than those used by FORTRAN.

The syntax of a global declaration is

(8) <global declaration> = global <token>,...,<token>;
|global <token>;
|global <signed integer><token>,...,<token>;
|global <signed integer><token>;
<signed integer> = <integer> | - <integer> .

Examples are:

global addroutine,x1,x2,addroutine_y;
global 3 optflag;
global -1 case_flag;

A name *nm* available in a given scope *ns* and declared global in that scope possesses a globality level, defined as follows: if the global declaration in which *nm* appears begins with a <signed integer> *k* , the value of *k* determines the globality level of *nm*. If such a <signed integer> is absent from the global declaration in which *nm*

appears, then the globality level of *nm* is (by default) equal to the level of the scope *ns*.

Suppose, for example, that the three global declarations shown above appear in the context

```
scope 2 library1;
      global addroutine,x1,x2,addroutine_y;
      global 3 optflag;
      global -1 case_flag;
      ...
```

Then *addroutine*, *x1*, *x2* and *addroutine_y* have globality level 2; *optflag* has globality level 3, and *case_flag* has globality level -1.

An item *nm* designated by a name available within a scope *ns* and having a given globality level *n* becomes known within every scope *ms* directly imbedded within *ns*, provided that the level of the scope *ms* does not exceed n. Moreover, if *nm* 'penetrates' into *ms* (i.e., becomes available via globality within *ms*), it has default globality level *n* within *ms*, and will therefore become known within all imbedded subscopes of *ms*, provided that the level of these subscopes does not exceed *n*. This global propagation of name availability will continue through a series of mutually imbedded scopes until either a scope of level exceeding *n* or a scope containing no subscopes is encountered. An item *nm* known within a namescope *ns* by the alias *x1* is known under the same alias *x1* within all scopes *ms* to which it is propagated through global declarations.

As already noted, each SETL variable will be *owned* by a particular procedure; when this procedure is entered, the current value of the variable will be stacked; the value will be unstacked on return from the procedure. The SETL conventions determining variable ownership are as follows. If a variable is known only within a single procedure (i.e., within the namescope which is coextensive with the procedure, or within several namescopes, all embedded with the procedure) it is owned by the procedure. If it is known in a body of text more extensive than a single procedure, it is owned by a nominal 'global system procedure' (and hence not stacked on entry to any procedure) unless this general default rule is over-

ridden by the presence of an owns declaration. Such declarations all have the following syntax:

(9) owns $routname_1$(varname$_1$, varname$_2$, ...),

$routname_2$(varname$_{k+1}$,varname$_{k+2}$,...), ...;

Here *routname*$_1$, *routname*$_2$, and so forth are tokens, possibly compound, which must designate items *i* which are subroutines or functions (that is, *i* must appear following either the keyword *define* or the keyword *definef*.) Moreover, *varname*$_1$, *varname*$_2$, etc. are tokens, possibly compound, which must designate variables.

Here is an example:

```
(10)    scope treerouts; global nodes, ℓ, r;
        owns walk(nodes);
        define walk(tree); nodes = nℓ; walkfrom(tree);
         return nodes;  end walk;
        define walkfrom(top); /* a recursive routine */
        top in nodes;
        if ℓ(top) is newt ne Ω then walkfrom(newt);
        if r(top) is newt ne Ω then walkfrom(newt);
        return;
        end walkfrom;
        ...
        end treerouts;
```

In this example, the set *nodes* is stacked on entry to the routine *walk*, but not on entry to the (recursive) routine *walkfrom*. This allows *walkfrom* to collect items in a set external to itself. The items *l* and *r* are not stacked on entry to either routine. The variable *newt* is stacked on each entry to *walkfrom* (though, as a matter of fact, this is not essential to the logic of the above programs).

Note as an exception to the above that label items, i.e., items designating labels (see below for details) are always owned by the nominal global 'default' procedure and hence never stacked.

We now turn to describe the SETL include declaration. This declaration can be used when a scope item *ms* (more precisely, a name designating a scope item) is known within a scope *ns* (for

example, *ms* may denote a sibling scope of *ns*); use of this declaration identifies one or more items known within *ms* with items known within *ns*.

In preparation for a discussion of the semantics of include statements, we discuss their syntax. An include statement has the form

(11) include <list>, <list>, ..., <list>;

or, if only one <list> occurs, the simpler form

include <list>;

the syntax of <list> is as follows:

(12) <list> = <aliased name>|<aliased name>(-<token>,...,<token>)
|<aliased name>(<list>,...,<list>)| <aliased name>*

<aliased name> = <token> | <token> [<token>] .

Suppose that an include declaration of the syntax (11)-(12) occurs within a namescope *ns*. As will be indicated in more detail below, '-' is used to indicate that all items *except* those designated by a list of tokens are to become known within *ns*; a parenthesized list without a '-' indicates that *precisely* the items listed become available within *ns*; while '*' is used to indicate that *every* item known within some other scope also becomes available within *ns*. Finally, square brackets are used to achieve user control over the local alias under which items become available.

The use of the include statement will be grasped most readily through examples. First consider the following include statement.

(13) include optimizer(routs3(output(xl)));

The semantic force of this declaration may be described as follows. We assume that the declaration appears in a namescope *ns* in which a scope item i_1 with alias *optimizer* is known. Within i_1, a scope item i_2 is assumed known under the name *routs3*. Similarly, within i_3 an item i_4 with alias *output* is known and is a scope item. Finally within i_4 an item i_5 is known with alias *xl*. Under thes assumptions, the declaration (13) causes the item i_5 to be made available within *ns* under the alias xl_output_routs3_optimizer.

Next consider the example (14) which uses more of the power of the include declaration.

```
(14)  include optimizer(routs1*,routs2(-flowtrace),
                                routs3(input*,output));
      include output(x1,x2);
```

Suppose that these include statements occur within a scope *ns*. Suppose also that the name *optimizer* is the alias of a scope item known in *ns*. An item i_1 known in *optimizer* as *routs1* is made available in *ns* under the alias *routs1_optimizer*. In addition, the '*' appearing in (12) signifies that all items known in *routs1* are to be included in *ns*. If *x* is the alias of an item in *routs1*, its alias in *ns* is *x_routs1_optimizer*. All of the items in *routs2*, less the item known therein as *flowtrace*, are propagated into *ns*; this is the semantic force of the '-' appearing in (12). *Input* denotes a scope item available in *routs3*. As indicated by the second '*' in (12), all of the items known in *input* including the scope item itself are propagated into *ns*. If y is the alias of an item in *input*, its alias in *ns* is y_input_routs3_optimizer. Next, an item with alias *output_routs3_optimizer* is included. This last item, which in accordance with our general conventions may be referenced simply as *output*, is in fact referenced in the second include statement contained in (14). This makes available items i_1 and i_2 which are referenced in *ns* by the aliases *x1_output* and *x2_output*.

The reader can verify that the effect of the two declarations (14) is the same as that of the following more complicated single statement.

```
(15)  include optimizer(routs1,routs2(-flowtrace),
                          routs3(input*,output(x1,x2)));
```

We can supply an additional example concerning the use of the sign '*' in an include declaration by making reference to the earlier example (10). The names known in the scope *treerouts* of (10) are *nodes*, *l*, *r*, *walk*, and *walkfrom*. By writing

```
include treerouts*;
```

in a scope *ns* we make all these quantities available under local aliases nodes_treerouts, l_treerouts, r_treerouts, walk_treerout etc. This allows the use of *walk* and *walkfrom* as routine names and will normally be used to identify the *l* of *treerouts* with a similarly named variable occurring in *ns*, etc.

The above examples do not illustrate the alias-modification feature provided by the syntax (and semantics) of the *include* statement. The use of this feature is shown in the following example:

```
(16)  include graphops(transitivity_routines
                                (connectedness[cr](flag1),
          strong_connectedness[ ](flag1[scflag],flag2)));
```

Suppose that (16) occurs within a namescope *ns*, and that the scope name *graphops* (more precisely, the scope item designated by this name) is available within *ns*. Then the include statement shown above makes available within *ns* items the identities of which are determined as if the brackets ('[ ]') were not present. The brackets determine the alias under which each item is known in *ns*. Specifically the item *i* whose alias is *flag1* in the scope designated as connectedness_transitivity_routines in (16) is aliased in *ns* as

```
flag1_cr_transitivity_routines_graphops;
```

this is because 'cr' appears in the brackets following 'connectedness' and is substituted for 'connectedness' when the alias of *i* is calculated. For much the same reason the items aliased as *flag1* and *flag2* in the scope strong_connectedness are aliased in *ns* as

```
scflag_transitivity_routines_graphops
```

and

```
flag2_transitivity_routines_graphops  .
```

The null string appearing in the brackets following 'strong_connectedness' in (16) is substituted for 'strong_connectedness'; two underbars coalesce to one. As above, these compound tokens can be abbreviated in *ns* as *scflag* and *flag2* so long as no ambiguity results.

The above remarks concerning the include and global features ovided in our name-scoping scheme should make the general use and action of these features reasonably plain. Additional details will be given below; the conventions which apply in logically marginal cases can be deduced from an examination of the name-scope routines themselves, SETL code for which is given later in the present manuscript.

Various additional semantic rules and restrictions govern the manner in which our namescoping rules apply to procedures, procedure arguments, store block arguments, labels in procedures, and to macros. Some of these rules are deliberately restrictive, and intended to avoid errors which might easily and inadvertently creep in if over-free use of our very general declarations, especially the include declaration, were allowed. We shall now present these rules, thereby bringing our account of the SETL namescoping conventions to a certain level of completeness.

1. By making items i_1, i_2, etc. occurring within the proper text of one namescope *ns* known within another namescope *ms*, the SETL namescoping scheme allows these items to be identified with items j_1, j_2 occurring within *ms*, and then recursively with items k_1, k_2 occurring within a third namescope *ms'*, etc. A first restriction on the use of the SETL namescoping conventions may be stated as follows. We require that no identification made in consequence of the transmission of items between namescopes lead to the identification of two distinct items both occurring within a single scope. (The explicit alias declaration described in a later section allows this restriction to be relaxed.)

As an example of this rule, note that the following text is illegal:

(17)
```
scope routl;
... u = 0; v = 1; ...
end routl;
scope rout; global y, routl;
scope rout2;
include u_routl[y]; ...
end rout2;
[continued]
```

```
scope rout3
include v_routl[y];  ...
end rout3;
end rout;
```

Indeed, the declarations occurring in (17) would imply that u.routl and v.routl were identical, and the rule just stated excludes such identifications.

2. An item

(18) $t.ns_1. \ldots .ns_k$

(cf. (4)) occurring in a scope statement of the form (1) is called a scope item. Similarly,

2'. An item (18) occurring as a procedure name in a *define* or *definef* statement is called a procedure item;

2". An item (18) occurring as a subroutine argument in a *define* or *definef* statement is called an argument item.

2"'. An item (18) occurring in the context

(19) ... (store t) ...

within an explicit store block (cf. the section *Supplementary Discussion of Generalized Assignments* below) is called a store block argument item.

2^{iv}. An item (18) occurring as a label, i.e., occurring in a context such as

...; t: ... ,

is called a label item.

2^{v}. An item (18) occurring as a macro name in a macro definition (see the following section for a discussion of the conventions applying to macros) is called a macro-name item.

3. An item (18) which is either a scope item, a procedure item, an argument item, a store block argument item, or a macro name item is said to be of definite initial designation. Other items in a total SETL text are said to be of indefinite initial designation.

A second restriction on the use of the SETL namescoping conventions may be stated as follows. We require that no identi cation made in consequence of the transmission of items between

namescopes identify two distinct items of definite initial signation. On the other hand, our rules do allow an item of definite initial designation to be identified with an item of indefinite designation, and do allow two items i_1 and i_2 of indefinite initial designation to be identified (provided, however, that these two items do not occur in the same namescope; cf. remark 1, above).

The rule just stated makes it impossible to identify an item designating a subroutine label with an item designating a subroutine or namescope, etc. Thus, for example, the following usage is illegal.

(20)
```
scope all;
scope ab;
scope a;
...
end a;
scope b;
...
end b;
end ab;
scope a;
...
end a;
end all;
```

Indeed, the text (20) would lead to the identification (within scope *ab*) of the items a.ab.all and a.all, which are distinct items both having definite initial designation (since both are scope items).

Note however that the following usage is legal:

(21)
```
        define f(x); ...
        x = 0;
labela: y = 0;
        end f;
        define g(x); include labela_f[y];
        end g;
```

Note that in the context (21) the variable y.g references precisely the same item as does labela.f. This rather artificia but nevertheless legal text makes the label-atom i designated by labela.f available within the subroutine g (which is also a name-scope). Note however that SETL does not allow direct transfer from g to this label. However, some other label atom might conceivably be tested within g for identity with i; other ways of using this sort of construction will appear in examples given below.

4. A procedure item (18) is also a valid SETL variable, whose value is initialized (at compile time) to the procedure atom *p* created by compiling the unique *define* (or *definef*) statement in the namescope nest $ns_1.ns_2. \ldots .ns_k$ within the compound token *t* appears as procedure name. Similarly, a label item (18) is also a valid SETL variable, whose value is initialized to the label atom which corresponds to the unique use of *t* as a label within the namescope nest $ns_1.ns_2. \ldots .ns_k$.

This rule validates the normal SETL use of procedures and labels. The following examples will illustrate other usages governed by this rule. First consider

(22)
```
scope routs; global forg;
definef f(x); ...; end f;
definef g(x); ...; end g;
define switch(x); forg = if x gt 0 then f else g;
return;
end switch;
end routs;
```

Suppose that in the presence of (22) we also have

(23)
```
scope more; include routs*; ...; end more;
```

Then *forg* and *switch* are available within *more*. Before a first call to *switch*, *forg* will have the value Ω. After calling switch(-1), forg(x) will return the same value as g(x); after a call to switch(+1), forg(x) will return the same value as f(x).

We give a similar example of the use of label-valued variable Consider

(24)
```
                scope example2; global lab, labzero, labone;
                definef ff(x); go to lab;
        labzero: return 0;
        labone:  return 1;
                end ff;
                define switch2(x);
                lab = if x gt 0 then labzero else labone;
                return;
                end switch2;
                end example2;
```

In this example, *ff* will return 0 immediately after a call to switch2(-1), and 1 immediately after a call to switch2(1).

5. Items (18) which are either scope items or macro-name items will be called passive items. We impose the restriction that no passive item, and no item which comes to have the same reference as a passive item, can be used as a SETL variable. Note for example that this rule makes the following usage illegal.

(25)
```
                scope yy; global yy;
                define f(x); ...
                yy = 0; ...
                end f;
                end yy;
```

Note on the other hand that the usage

(26)
```
                scope yy; global yy;
                scope inner; ...
                scope moreinner;
                ...
                end moreinner;
                end inner;
                end yy;
```

is legal. This usage makes the scope *yy* known within the scope *moreinner*; the same effect would be obtained if the declaration

(27)
```
                include yy_inner;
```

e present in *moreinner*.

6. As has been noted, each procedure body is at the same time a namescope (of level zero). Each procedure item is therefore also a scope item. However, we deliberately forbid either a procedure item or an item which comes to have the same reference as a procedure item to appear in include declarations, except in terminal position. That is, such an item may not be followed in an include declaration either by a parenthesized list of <token>s, by a parenthesized list of <token>s preceded by the sign '-', or by an asterisk. Note for example that this rule makes the following usage illegal:

(28)
```
definef f(x); ...
y = y+1; ...
end f;
definef g(x); include f(y); ...
end g(x);
```

The referencing effect that (28) would attain (were it not illegal) can be achieved by writing the somewhat clumsier

(29)
```
scope auxil; global y; owns f(y);
definef f(x); ...
y = y+1; ...
end f;
end auxil;
definef g(x); include auxil(y);
end g(x);
```

The restriction which has just been stated is imposed so as to ensure that no variable used in a procedure can be referenced remotely unless a visible declaration of intent to do so appears in some relationship of physical proximity to the body of the procedure. This restriction is mild; to abandon it would be to invite trouble.

The SETL concept of ownership of variables, which determines the manner in which variable-value references will be changed by recursive subroutine calls and returns, was mentioned occasionally in the preceding pages. We shall now give additional detail concerning the semantic conventions relevant to this notion.

Every variable in a SETL program is 'owned' by some procedure, e., is treated at SETL's basic level of semantic interpretation as the k-th variable of some j-th subroutine. It is possible that j should be zero; j = 0 corresponds to a nominal 'default' procedure, which owns certain classes of variables (such as those corresponding to labels and procedure names) which are never either stacked or unstacked. In regard to stacking and unstacking, subroutine arguments are treated in much the same way as other variables: if the j-th subroutine has m arguments, these will be its 1st,2nd,...,mth variables. That additional argument,often hidden, parameter of functions which may become explicit as a store block parameter if the subroutine is called in sinister mode (cf. the section *Supplementary Discussion of Generalized Assignments* below) is its $m+1^{st}$ variable.

When a procedure is entered, values are established for its arguments, following which one begins to interpret the code constituting the subroutine body. If the procedure is re-entered recursively, then the value of each variable which it owns is stacked before new values are established for its arguments. At this same time, the value Ω is established for each non-argument variable owned by the procedure. When return is made from the procedure, an unstacking action returns all variables owned by the procedure to their previous condition.

Significantly different rules apply to 'base-level' and to 'recursive' entry to a procedure. An invocation counter, initially set to 0, is maintained for each procedure; this counter is incremented by 1 each time the procedure is entered, and decremented by 1 each time return is made from the procedure. A base-level entry to the procedure is one which moves its invocation counter from 0 to 1; other entries are said to be recursive. When base-level entry is made to a procedure, values are established for its arguments, but all other variables owned by the procedure, rather than being set to Ω, retain their pre-entry values. These will generally be the values which they had at the last prior return from the same procedure. Note also that, at the end of mpilation, but immediately before execution begins, the value or each subroutine item will be initialized to an appropriate

subroutine atom, and the value of each label item will be initialized to an appropriate label atom. The initial value Ω will be established for all other variables.

Suppose that an item i in a procedure f has the same reference as an item j in a procedure g. Suppose also that i is owned by g, so that the value which it designates changes as recursive entries to and returns from f are made. Then j always references the current value of i; so that the value designated by j also changes as calls to and returns from f are made. The following example, which the reader is asked to ponder, illustrates this point.

(30)
```
global y,z; owns proc1(y);
define proc1;
proc2; y = 1; proc2;
if z eq Ω then z = 1; y = y+1; proc1;
   else return; end if;
y = y+1; proc2; return;
end proc1;
define proc2; print y;
end proc2;
proc2; proc1;
```

The code (30) will cause *proc2* to be entered six times; thus six values of y will be printed. On the first entry to *proc2*, y will have its immediate post-compilation value Ω; thus Ω will be printed. This value will be printed again when *proc2* is entered immediately after a base level call to *proc1*. When *proc2* is next called, y will have been changed to 1, and 1 will be printed. Next, *proc1* will call itself recursively. On the recursive entry to *proc1*, the value Ω will be established for y, and Ω will be printed. Following this, 1 will be printed. Recursive return from *proc1* will then restore y to its previous value of 2, which will be incremented once more before *proc2* is again entered, causing 3 to be printed. All in all, the output sequence produced by the code (30) is

Ω, Ω, 1, Ω, 1, 3 .

7. Macros

No language is ideally adapted to all possible application areas, and for this reason it is desirable for languages to be modifiable and extensible. To specify maximally powerful extension mechanisms is a complex task. We shall evade this task at the present point, and shall in fact confine ourselves to describing a relatively simple SETL macro-processor feature. Note that macroprocessors are relatively straightforward mechanisms allowing a programming language to be modestly 'perturbed' in ways which a user can find quite convenient. While generally not permitting the extensive linguistic variation made possible by more elaborate syntax modification schemes, they are generally easier to use than full-fledged extensibility schemes: a relatively light tool well adapted to light usage.

The SETL macro-system to be described will basically be conventional and straightforward. However, some complications will arise because of our desire to have the macroprocessor conform to the namescoping conventions that have just been described. Note in this connection that many of the general remarks concerning names and namescoping made at the beginning of the preceding section also apply to the use of names as the names of macros. In dealing with large masses of text, it is important that the scope within which a name has macro-name status be limited. If this is not the case, macro names will steadily accumulate, and, given a sufficiently large mass of program text containing macros, will become difficult to manage. The scopes within which one desires to use a given library of macro definitions will not always be physically contiguous; thus a way of transmitting macros by something akin to an include declaration is desirable. Finally, since program clarity should be the exclusive factor controlling the order in which one arranges the parts of a total text, it is best to avoid restrictions which force macro definitions to appear in some fixed physical relationship to invocations of the macros which they define.

The SETL macro-scheme which we now begin to describe satisfies the first two of these desiderata, but not the third. This scheme allows names to be used as macro names within specified namescopes,

and to be transmitted between scopes, in the same fashion as other names, by global and by include declarations. However, in order to avoid the problems of definition and of implementation which would otherwise result, we require that macro-definitions physically precede invocations of the macros they define. The precise conventions which apply will be stated below.

Before entering into a detailed discussion of these name-scope related issues, we describe the more basic rules which apply to macros and macro-invocations within a single namescope.

A macro-definition has one of the following forms

(1a) macro name; text endm name;
(1b) macro name(arg_1,...,arg_k); text endm name;
(1c) macro name(arg_1,...,arg_k;$genarg_1$,...,$genarg_m$); text endm name;
(1d) macro name(;$genarg_1$,...,$genarg_m$); text endm name;

In each of the definitions (1a)-(1d), *name* is a (possibly compound) token, which the definition designates as a macro name; *text* is any string of tokens, constituting the so-called body of the macro. Note however that in (1a)-(1d) *text* cannot contain the token

(2) name

except under restrictions to be stated shortly.

A definition of the form (1b) involves user-supplied macro arguments; (1c) involves both macro arguments and generated macroarguments. Definitions of the form (1a) involve neither arguments nor generated arguments, and are consequently simplest; we shall explain the argumentless macros of this form before going on to discuss the somewhat more complicated cases (1b)-(1d).

The definition (1a) causes each following occurrence of *name* (other than those preceded by one of the keywords macro or end) to be replaced by the *text* which appears in (1a). In the presence of several macrodefinitions, repeated substitution will be carried out. Thus, for example, every occurrence of *a* following the definitions

(3) macro a; b c; endm a;
macro b; d e; endm b;
macro c; f g; endm c;

is replaced by an instance of the four-token sequence

(4) d e f g .

The body one one macro can contain the definition of another. In this case, the imbedded macro-definition becomes active when the macro containing it is invoked. For example, following the definition

(5) macro a; macro b; c d; endm b; endm a;

the token sequence b a b is replaced by the sequence b c d. Note that b is only replaced by c d after the definition (5) is invoked (by an occurrence of *a*).

Note in connection with all of this that a macro definition is not allowed to cross a namescope boundary, except when propagated in the manner explained below.

Macros with arguments, having definitions of the form (1b), allow additional flexibility. If *name* is a macro-name with the definition (1b), then it is invoked by the occurrence of any token sequence of the form

(6) $\text{name}(\text{text}_1,\ldots,\text{text}_k)$.

In (6), $text_j$ denotes any sequence of tokens not containing a comma which is not included within parentheses. An invocation (6) of the macro (1b) is replaced by an occurrence of the *text* body of (1b), but within this *text* each occurrence of the j-th argument token arg_1 is replaced by an occurrence of the corresponding j^{th} actual argument $text_j$ appearing in the invocation (6). The text of a macro with arguments may contain imbedded macro-definitions; these definitions become active when the macro is invoked. Macro expansion is recursive, and outside-in.

A macro (1c) with generated arguments is invoked in precisely the same way as the corresponding macro (1b), i.e., has the invocation (6). The effect of the generated macroarguments appearing in :) may be described as follows. Immediately prior to the expansion

of the macro-invocation (6), a set of m unique names $n_1,\ldots,n_m$ of a reserved form are generated by the macroprocessor. These are th treated during macro-expansion as additional macro arguments; that is, each occurrence of the j-th generated macroargument token $genarg_j$ is replaced during macro-expansion by an occurrence of the freshly generated name n_j. As is well known, this feature is convenient for generating text required to contain unique variable names, labels, etc.

Formula (1d) shows the manner in which a macro with several generated macroarguments but with no user-supplied arguments is defined.

Note that the second of two definitions of a macro with a given name replaces the first. That is, expansion will be made according to the first definition only until the second definition occurs, after which expansion will be made according to the second definition.

A name may be dropped from macro-status by writing the degenerate definition

(7) macro name endm;

In many cases tuples of fixed length will be used to store some group of object attributes; in such cases the particular order of components is irrelevant, though of course some order must be imposed since each component has a significance distinct from all the others. In such situations, it is desirable to avoid numerical reference to particular components, since the use of numerical references infests a text with semantically meaningless encodings, always a thing best avoided for clarity and for modifiability. Of course, the use of macros can alleviate this situation. If, e.g., we deal with objects represented by triples whose successive components represent size, weight, and price, then, instead of systematically writing obj(1), obj(2), and obj(3) for these three attributes we can introduce the following three macros and write size(obj), weight(obj), and price(obj):

(8)
```
macro size(x);   x(1) endm size;
macro weight(x); x(2) endm weight;
macro price(x);  x(3) endm price;
```

However, since situations of this kind are quite common, and since e avoidance of numerical references is much to be encouraged, we provide an easy-to-use special macro form replacing patterns such as (8). This is the definition form

(9) macro $name_1$,$name_2$,...,$name_k$; endm;

The form (9) has precisely the same force as

(10)
```
macro name1(x); x(1) endm name1;
macro name2(x); x(2) endm name2;
                ...
macro namek(x); x(k) endm namek;
```

We now turn to describe the conventions which relate macros to namescopes and govern the transmission of macros between scopes. Macro names may be declared global and may appear within include statements. (However, a macro-name may appear in an include statement only in terminal position (cf. the preceding section, paragraph immediately preceding formula (28)). That is, such an item may not be followed in an include declaration either by a parenthesized list of <token>s, a parenthesized list of <token>s preceded by the sign '-', or by an asterisk.) Macro-definitions are processed and macro-expansion performed during an initial, *advancing* pass over SETL source text; in this pass, scope boundaries are established, the items forming part of the proper text of each namescope collected, and the processing of global and include statements begins.

If a macro-name is declared global within a namescope *ns*, it is propagated into every scope physically included within *ns*, and is then treated as a macro from the point at which its macro-definition occurs, and thereafter either to the end of the scope *ns* or to the next following macro-redefinition of the macro name. The comments in the following example show some of the implications of this statement.

(11)
```
scope withamacro; global mname;
mname=mname+1; /* this is  legal, since
 mname has not yet been defined to be
 a macro. no expansion yet */
```

[continued]

```
macro mname(x); x=x+1 endm mname;
mname(t); /* expands as t=t+1; */
scope inner;
mname(n); /* expands as n=n+1; */
macro mname(x); x=x-1 endm mname;
mname(v); /* expands as v=v-1 */
macro mname endm; /* dropping mname
 from macro status */
mname = mname+1; /* this is again legal,
 since mname has been dropped from macro
 status, and is not expanded */
end inner;
mname = mname+1; /* still legal;
 mname is still not a macro */
end withamacro;
```

If a macro-name *mn* appears in an include statement within a namescope *ns*, it is propagated into *ns*, and is treated as a macro from the point at which the include statement occurs, and thereafter either throughout the proper text of *ns*, or to the next following macro-redefinition of the macro name in *ns*. Note however that for *mn* to become known as a macro name the first line of the namescope *ns'* from which the include statement propagates *mn* into *ns* must physically precede the end of *ns*, and *mn* must be known as a macro name within *ns'*; its macro-definition must also precede the end of *ns* physically. Comments in the following extended example illustrate this rule.

(12)
```
scope early; include next(mname);
mname=mname+1; /* this is legal, since the
 scope next within which mname is known as
 a macro has not yet been encountered */
end early;
scope next;
macro mname(x); x = x+1 endm mname;
mname(t); /* expands as t = t+1; */
end next;
scope later; include next(mname);
mname(n); /* expands as n=n+1 */
```

[continued]

```
/* now we redefine mname */
macro mname(x); x=x-1 endm mname;
mname(v); /* expands as v=v-1 */
end later;
scope latest; include next(mname),
 later(mname[newn]);
mname(w); /* expands as w=w+1; */
newm(z); /* expands as z=z-1; */
end latest;
end early;
```

The last few lines of the preceding example illustrate the following rules. The final macro-status within a namescope *ns* of a name *mn* is that macro-definition (if any) which applies at the moment that the scope-closing end statement is encountered. If *mn* is transmitted from *ns* via an include statement to another namescope *ns"* (which follows *ns* physically) then within *ns"*, *mn* will have what was its final macro-status within *ns*. As is shown by the last few lines of the preceding example, this rule applies uniformly.

8. Input and output.

Input/output conventions sufficiently substantial to allow SETL to make use of externally stored files will be described later. Here, however, we shall describe only a rather rudimentary set of input/output facilities, beginning with a basic method for handling character strings in an essentially 'unformatted' way.

The allowable characters in a character string are all the normal members of a standard character set, plus one additional character designated er (end record). If s is an ordered pair <st,n> consisting of a character string *st* and an integer n referencing one of its characters, the system function *record* may be called from within any expression, in the form

(1) record s .

The value v of this function is the segment of the string *st*, beginning at its n-th character, and including all characters of s up to but not including the first occurrence of the character er; when

record is called, it increments the second component, n, of its argument by # v + 1.

If the n-th character of s is er, then *record* returns the value nulc and increments n by 1. If n exceeds the length of the string s, then *record* returns the value Ω, and does not increment its argument n; this fact can be used to perform the SETL analog of the normal 'end-of-file' test.

All this describes an action appropriate for an input reader. Note in connection with the above that the ordinary analog in SETL of a 'file' is a pair s = <st,n> consisting of a character string and an integer referencing one of its characters. That is, the basic SETL file system is provided simply by allowing very long character strings to reside on an external medium, and by ensuring that both the function (1), and the primitive which appends one character string to another, are supported in a reasonably efficient manner.

The function

(2) open str

is used to link SETL to an operating system for input/output purposes. It acts as follows: 'str' is a character string, giving the name under which some possibly very long character string *s* is known to the operating system. The operation (2) requests this string from the operating system; morever, the value of the expression (2) is the string *s* itself. Thus, by writing

(3) x = open str;

or perhaps

(4) y = <open str,1>;

we make the string *s* available within SETL as the value of the variable x (or as the body of the 'file valued' variable y).

The body of a file is returned to the operating system by writing

(5) close(x,str);

Here, *x* is a SETL variable whose value is the body *s* of the file to be returned, and *str* is a character string, giving the name under which *s* is to be known to the operating system.

If a character string is returned to the operating system as .nt output, the following conventions will apply. Each occurrence in the output stream of the character er will terminate the current print line with a period, followed by as many blanks as are necessary to fill out the line. Lines not containing an occurrence of the character er will always be terminated with a . blank, followed by a period. This convention allows character strings of arbitrary length to be transmitted to the output medium, and to be represented there unambiguously.

Next, we describe two SETL statements giving a rudimentary formatted input/output facility for use in connection with standard form input and output files. These have the form

(6) f print $expr_1$,$expr_2$,...,$expr_n$,

and

(7) f read $name_1$,$name_2$,...,$name_n$;

respectively, where f is an expression having as value an ordered pair <st,n> consisting of a character string and an integer. If in (6) or (7) the prefix f is omitted, the variable name *input* is understood in case (7); and the variable name *output* is understood in case (6).

The form in which a set will be printed is determined by the following recursive conventions. An integer will appear in decimal form, possibly preceded by a minus sign.

A character string will appear enclosed within quote marks in its normal external form, quote marks themselves being represented by double quotes. Bit strings will appear either in "binary" forms such as 01100101...01B, in octal form 0770070, or in a combined "binary-octal" form, in which a binary prefix precedes the letter B and an octal suffix follows it, the total bit-string being the concatenation of these two separately represented parts. Note for example that the strings 10111000B and 10B70 are identicaly; and that either form may be used in a SETL program to represent a bit-string constant. Real numbers will appear either in a decimal form such as 90., 0.9, or .99, or in an exponential form such as -5.

[f a1,...,an are the elements of a set s, and r1,...,rn are the printed representatives of a1,...,an respectively, then the set

{a1,...,an}

will generally appear printed as the character string

{r1,...,rn}

a similar convention applying to n-tuples. These simple recursive rules will be used as long as the length of the character strings it produces does not exceed two printed lines, and the level of parenthesis nesting needed within these character strings does not exceed 4. When these limits are exceeded, subsets or tuples of a composite structure which is to be printed will be assigned abbreviating designators consisting of an integer followed by a colon, and the significance of such abbreviations will be indicated on separate printed lines, indentation being used appropriately to improve the readability of the resulting text. Thus, for example, a set that might have been printed as

(8) {{{{5,10,15,<20,[21,22,23,{{24,<25,8>},9}]>,3}}}}

will actually appear as

(9) {{{{5,10,15,1:,3}}}}

1: <20,[21,22,23,{{24,2:},9]>

2: <25,8> .

The whole external representation of a set will be terminated by a slash, i.e., by the sign / . This is a symbol that is not allowed, unquoted, in a file to be read, and is used during reads to check for possible misparenthesization or other malformation of read input. The SETL read statement (7) will accept, from a specified file, sets represented in the manner illustrated by (8), and convert them into the abstract structures which they represent, assigning the set thereby obtained as the value of the variable *name* occurring in (7). The number of characters of the *input* string digested during such a read operation is determined by the following rule: blank characters, up to a first non-blank character, will be ignored. If the first nonblank character is either <, or { a 'balanced parenthesis' section of input, never including an occurrence of the sign /, will be digested. If

the first nonblank character is /, it will be ignored. If the
rst nonblank character is anything else (but not a comma, >,
or }) it will be read, and returned as the result of the read
operation. Blanks (except quoted blanks) will be ignored, and
single characters er (but not double characters er) will be
ignored also.

Item 15. A LIBRARY OF EXAMPLES SHOWING THE USE OF SETL

1. Algorithms for lists and trees.

A unilateral list may be regarded as a set of items, supplemented by a function *next(item)* such that next(item) = Ω for the last item. In addition, a pointer *first* locating the first item in the list must be given. The basic list operations are insertion after a given position and deletion of the next item after a given position. Note that the set of items in the list is the union of the domain and range of *next*, so that the list is completely specified once *next* is given. The following insertion and deletion routines insert or delete the first element if called with Ω as parameter; otherwise they insert or delete after whatever list position their parameter specifies.

```
define item insafter prev; /* next and first are assumed to be global*
if prev ne Ω then
   <next(item),next(prev)> = <next(prev),item>;
else
   <next(item),first> = <first,item>;
end if;
return;
end insafter;
define delafter item; /* next and first are assumed to be global */
if item ne Ω then
   nx = next(item);  if nx eq Ω then return;;
   next(item) = next(nx);  next(nx) = Ω;
else
   oldfirst=first; first=next(first); next(oldfirst)= Ω;
end if;
return;
end delafter;
```

A bilateral, circularly linked list may be regarded as a set of items with functions next(item), prev(item) defining the successor and predecessor of a given item; the last item is considered to be the predecessor of the first item, and the first

the successor of the last. The first item on the list is designated by *first*. Note that the three objects *next*, *prev*, and *first* together specify the list. The basic operations are insertion of an item after a given position and deletion of a given item. These procedures may be written as follows.

```
/* in the two following routines next, prev, and first
   are assumed to be global */
define  item insbilat prec;
pre = prec; /* to avoid changing argument */
if pre eq Ω then /* empty list or insertion at head of list */
   if first eq Ω then /* initialize empty list */
      <next(item),prev(item),first>=<item,item,item>; return;
   else <pre,first> = <prev(first);item>;; /* insertion at head */
end if pre; /* now  pre indicates the point of insertion */
<next(item), next(pre), prev(item), prev(next(pre))> =
        <next(pre), item, pre, item>;
return;
end insbilat;

define delbilat item;
<next(prev(item)), prev(next(item))> =
       <next(item), prev(item)>;
if item eq first then
       first = if next(item) is x ne item then x else Ω;
end if;

next(item) = Ω;  prev(item) = Ω;
return;
end delbilat;
```

A binary tree is a set of nodes and two descendant functions r and ℓ (right and left descendants); a given top node *ntop* must also be specified. The tree is then entirely defined by these two functions and the specified top node. It is often necessary to traverse a tree in some standard order. We take as an example left-top-right traversal order, and generate the sequence or nodes in the order traversed. Note that *seq* must be *owned* by some routine other than *traverse*.

```
/* in the following seq, l, and r are assumed to be global */
seq=nult;  traverse ntop;
define  traverse top;
if top eq Ω then return;;
traverse l(top);  seq(#seq+1) = top;  traverse r(top);
return;
end traverse;
```

An ordered tree is a set of nodes with a descendant function desc(node,j) defined for j in some finite (possibly null) range. Ordered and binary trees stand in an interesting 1-1 relationship. Given an ordered tree, it may be converted to a binary tree by designating the first descendant of a node N as N's left descendant; and by designating desc(n,j+1) as the right descendant of desc(n,j). In SETL:

```
l = {<n,desc(n,1)>, n ∈ tree | desc(n,1) ne Ω};
r = {<desc(n,j),desc(n,j+1)>, n ∈ tree, 1 ≤ j < #desc{n}};
```

To invert the above transformation, one takes a binary tree and makes (n) and the successive right descendants of (n) in a binary tree as the successive descendants of n in the corresponding ordered tree. In SETL we have:

```
desc = nl;
(∀n ∈ tree)
   k = 1;  d = l(n);
   (while d ne Ω doing k = k+1; d = r(d);)
      desc(n,k) = d;;
end ∀n;
```

To form an isomorphic copy of a binary tree, the following procedure may be used. Note that a similar procedure will serve to form an isomorphic copy of any structured object:

```
copy = {<n, newat>, n ∈ tree}
l = l + {<copy(n),copy(l(n))>, n ∈ tree | l(n) ne Ω};
r = r + {<copy(n),copy(r(n))>, n ∈ tree | r(n) ne Ω};
```

A *threaded tree* is represented by a set *tree* on which two nctions r(node) and ℓ(node) are defined, each for all but one node. The values of each of these functions are ordered pairs. We have r(node) = <node',flag>, where *node'* is either the right descendant of *node* or its successor in left-top-right traversal order, depending on whether flag eq t or flag eq f. Similarly, ℓ(node) = <node',flag>, where *node'* is either the left descendant or the traversal-order predecessor of *node*.

The following example, showing flagged pointers as dotted arrows, illustrates the notion of a threaded tree.

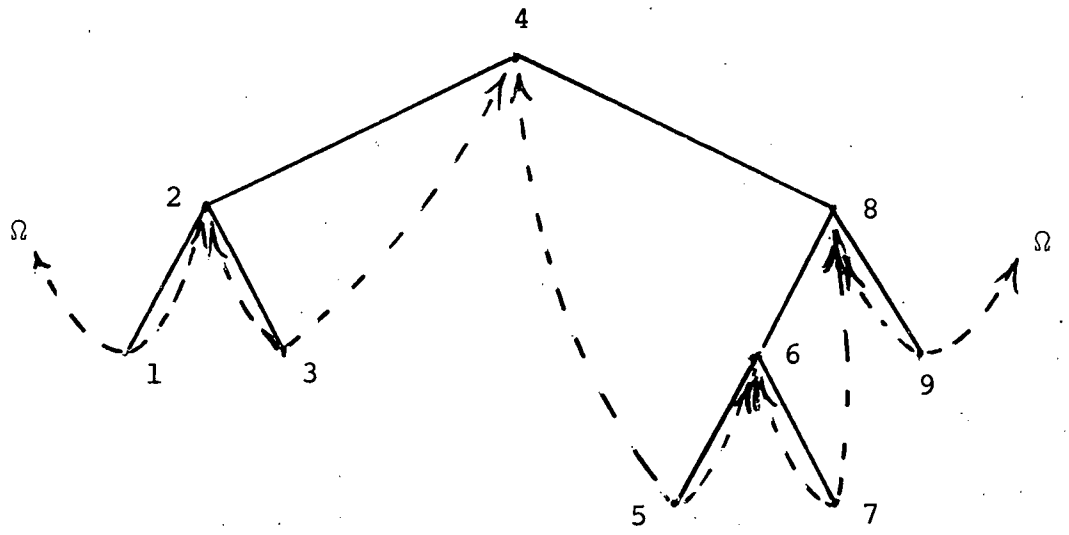


Fig. 1. A threaded tree

To traverse a threaded tree in left-top-right order we may use the following code:

```
seq=nult;  node = top;
flow                                  isldef?
                  islflag?                          add+
            down+     add+                        isdone?
            isldef,  isdone,                  quit,   down+
                                                      isrflag?
                                                isldef,   add+
                                                          isdone;
```

[continued]

```
isldef  := ℓ(node) is desc ne Ω;
islflag  :=desc(2) eq t;
down: node = hd desc;
add:  seq(#seq+1) = node;
isdone := r(node) is desc eq Ω;
isrflag := desc(2) eq t;                    end flow;
```

The following code inserts an element into a threaded tree, as the right descendant of a node *nod*:

```
define  elt putright nod;
/*  l and r are assumed to be global */
<r(elt),ℓ(elt), r(nod)> = <r(nod),<nod,f>,<elt,t>>;
if r(elt)(2) /* so that elt has an actual right descendant */ then
   desc = r(elt)(1); /* now descend to the left to repair
                                          the thread */
   (while ℓ(desc)(2)) desc = ℓ(desc);;
   ℓ(desc) = <elt,f>;  /* elt is the traversal-order
                                          predecessor of desc */
end if;
return;
end putright;
```

These processes are illustrated by the following figure, which shows the changes necessary to insert a node x as the right descendant of the fourth node in the threaded tree of Fig. 1. Only changed 'thread pointers' are shown.

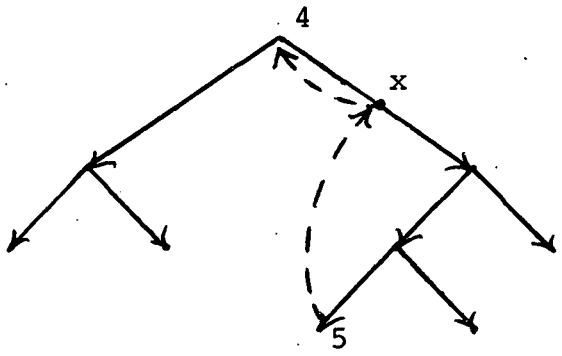


Fig. 2. Adding a left descendant in a threaded tree.

To thread an unthreaded tree, we first let *seq* be its nodes in ft-node-right traversal order, as defined by a previous algorithm. Then we use the following straightforward code, in which *tree* denotes the set of all nodes in the tree:

```
suc={<seq(n),seq(n+1)>, 1 ≤ n < #seq};
pred = {<x(2),x(1)>, x ∈ suc};
(∀n∈ tree) ℓ(n) = if ℓ(n) ne Ω then <ℓ(n),t>
  else if pred(n) ne Ω then <pred(n),f> else Ω;
     r(n) = if r(n) ne Ω then <r(n),t>
  else if suc(n) ne Ω then <suc(n),f> else Ω;
end ∀n;
```

It is simpler to convert a threaded tree to an unthreaded one, as follows:

```
(∀n ∈ tree) ℓ(n) = if(if ℓ(n) is thdesc eq Ω then f else thdesc(2))
                   then thdesc(1) else Ω;
            r(n) = if(if r(n) is thdesc eq Ω then f else thdesc(2))
                   then thdesc(1) else Ω;
end ∀n;
```

SETL cannot be used to express machine level optimizations. However, it can be used to express optimizations at an "abstract" or "algorithmic" level. Here, for example, is a parsimonious method, due to Schorr and Waite, for traversing a binary tree; in distinction to the methods given earlier, it avoids the use of a recursion stack. The idea is this: as one descends down a chain of branches to traverse the tree, one reverses the pointers, to get a chain of pointers allowing subsequent ascent. During ascent, the pointers are repaired. We mark those nodes n such that r(n) is the parent of n; in a machine-level implementation, at most one bit is needed for this mark.

```
seq = nult;  mark = nℓ;  node = top;  par = Ω;
flow                          isldesc?
                        dleft+              notenode+
                        isldesc,            isrdesc?
                                      dright+          hasparent?
                                      isldesc,   ismarked?     done,
                                            upright+  upleft+
                                           hasparent, notenode+
                                                      isrdesc;
isldesc: = ℓ(node) ne Ω;
dleft: <ℓ(node),node,par> = <par,ℓ(node),node>;
notenode: seq(#seq+1) = node;
isrdesc: = r(node) ne Ω;
dright: <r(node),node,par> = <par,r(node),node>;
        mark(par) = t;
hasparent := par ne Ω;
ismarked := mark(par) ne Ω;
upright: <node,r(par),par> = <par,node,r(par)>;
upleft: <node,ℓ(par),par> = <par,node,ℓ(par)>;
                                        end flow;
<done:> ...
```

2. A lexical scanner algorithm.

In the following pages we will use SETL to describe a number of algorithmic processes basic to the compilation of programming languages. We begin by describing a class of *lexical scanners*. These are programs, normally belonging to the very first stages of a compilation process, which accept an input string and break it up into separate *tokens*, i.e., strings of one or more characters representing words of a language. As it is calculated, each token is classified according to type (e.g., name, integer, character constant, boolean constant, etc.). Basic input conversions, as for example the conversion of a character string representing an integer to the internal form of the integer, may also be performed during lexical scan.

Because of its significant influence on the overall efficiency of the first stages of processing, one normally desires a lexical scanner to be quite fast. For this reason, lexical scanning is customarily performed by a programmed *finite-state automaton* which, driven by an incoming sequence of characters, undergoes a sequence of state transitions until a 'token end' state is reached; then any necessary conversions are performed and a token is emitted. We shall describe a lexical analyzer of this kind. The following background facts should be borne in mind.

i. The states of the automaton correspond to states of uncertainty concerning the nature of the token being constructed. When a new token is started, the scanner is in a state of complete uncertainty; this state is called *nxt* in the formal algorithm below. As characters are received, the state of uncertainty will change, always diminishing; when a token-end state is reached, the type of the token is entirely known, and is determined by the final condition of the automaton.

ii. The whole alphabet of characters belonging to a language may for the purposes of lexical scanning be regarded as consisting of a relatively small number of character classes (e.g. alphabetics, numerics, separators, alphabetics having special significance, etc). A function *type(character)* is therefore employed by the lexical

nner, which uses the value of this function and its own *state*

to find an *action table entry* which describes the action to be taken next. In the following algorithm, allowed standard actions are as follows:

a. *end* - end the present token without adding any additional characters to it, and return a triple defining the token itself and its lexical type.

b. *cont* (continue) - add the current character to the token under construction, and advance to the next input character;

c. *skip* - advance to next input character;

d. *go* (change state) - change to a specified state and then continue as in (b);

e. *do* (perform auxiliary process) - perform a specified sequence of operations including the execution of auxiliary code blocks supplied by the programmer. This code may examine and modify the token under construction, the state of the finite state automaton, the *action* parameter (see below) which the lexical scanner uses, any of its pointers, etc. If this code is some sort of conversion routine, it may supply token-associated data to the lexical scanner. In the algorithm which follows, auxiliary routines to be executed are all taken to be part of a common programmer-supplied *auxiliary process package* called *rpak*.

The lexical scanner routine which follows is called nextoken. It is written as a function which when called will break one additional token out of a given character string, returning as function value this token and its lexical type.

The forms assumed for the action table entries used in nextoken are as follows.

An entry may be:

aa. one of the keywords *end*, *skip*, and *cont*; or

bb. an ordered pair <*go*,*statename*>, where *statename* is the name of a lexical analyzer state; or

cc. an n-tuple <*do*,*routname*,...>, where *routname* is the name of an auxiliary routine, and where subsequent components are either *end*, *cont*, *go* followed by *statename*, or *do* followed by *routname*.

The detailed form of our lexical analyzer is as follows.

```
definef nextoken;
/* the tables 'type' and 'table', and the code block 'rpak'
     are produced by the routine 'setup' given later in
     this section */
initially tokbegin = 1;
<nxt, end, go, skip, cont, do> = <'nxt','end','go','skip',
     'cont','do'>;
/* we assume for simplicity (but somewhat unrealistically)
     that the entire input  string is read in before lexical
     scan begins.  Note that the input string is broken into
     successive records, and terminated by a double end record */
/* we assume that the file input has been opened for reading
     by a prior instruction */
/* do read-in operation until double end record */
this = Ω;  cstring = nulc;
(while this ne nulc)
     this = record(input);
     cstring = cstring + this + er;
end while;
end initially;

state = nxt; curpointer = tokbegin-1; data = Ω; token = nulc;
loop: curpointer=curpointer+1;
  action = table(state, type(cstring(curpointer)));
switch: go to {<end,endc>,<go,goc>,<skip,loop>,<cont,contc>,
       <do,doc>} (if(type action) eq tupl then
         hd action else action);
goc:    state = action(2);
contc:  token = token + cstring(curpointer); go to loop;
endc:   tokbegin = curpointer; return<state,token>;
doc:    rout = action(2); action = action(3:); rpak(rout);
        go to if action eq Ω then loop else switch;
end nextoken;
```

This routine is simple enough; as we shall soon see, the routine *setup* which supplies the tables needed by *nextoken* is rather more complex. Concerning *setup*, we have made the following assumptions.

i. It finds the information needed to define the character type function *type*, the action table *table*, and the comprehensive package *rpak* of auxiliary processes at the head of the string *input*, all represented in a form which can be read using the SETL *read* statement. This information is read, checked for accuracy, and converted appropriately to produce the required tables.

ii. In more detail, the information supplied to *setup* is as follows. First, a string including every valid character other than er is given. For SETL,this would be

'abcdefghijkℓmnopqrstuvwxyz0123456789()[]{},*+-/=?"<>≤≥Ω#∀∃:|.'

Note that a double quote within quote marks represents a quoted single quote.

After this string, there follows a tuple

<ctypel,...,ctypen>

in external form (i.e., in a form suitable for ingestion by a *read* statement), defining the full collection of character types with which the lexical scanner will be concerned.

Suppose, for example, that we consider a hypothetical language lexically somewhat like FORTRAN,in which the allowed lexical types are as follows.

a. Integer: any sequence of digits, embedded blanks allowed.

b. Real number: an integer, followed by a decimal point, and optionally followed by a second integer.

c. Name: any string of nonspecial characters beginning with an alphabetic; no embedded blanks allowed.

d. Special character: any character other than blank or period.

e. Period delimited operator: any string of nonspecial characters, beginning with an alphabetic, containing no blanks, and delimited fore-and-aft with a period. Examples would be: .ge. , .shift. .

f. Hollerith constant: any number of digits, followed by the letter h, followed by an arbitrary character string of the length specified by those digits. An example would be: 5hhocha. We suppose for simplicity that *er* functions as an end-of-statement signal, no continuation-card feature being provided.

For such a language, the relevant character types could be declared to *setup* in the following form

(1) <a,h,'l',+,.,bℓ,er> .

Here 'a' is used to designate the type of the typical alphabetic (which is to say alphabetics not equal to h, since h plays a special role in hollerith constants); l to designate the type of a numeric; + to designate the type of a special symbol other than '.', etc. We use 'er' to designate the type of an end record symbol, 'bℓ' to designate the type of a blank.

iii. Next there follows a set of tuples, which together define the lexical type of every possible character. These n-tuples have the form

$$<type,cstring_1,cstring_2,\ldots>$$

where *type* is a previously declared lexical type, and $cstring_j$ is a string of characters, all of which are declared to have this type. The special character string 'er' is however reserved to represent the SETL end record character *er*.

In the case of our hypothetical FORTRAN-like language (which however is assumed to include the full SETL character set) we would have

(2) {<a, abcdefgijklmnopqrstuvwxyz>,<'l','0123456789'>,
<+,'?()[]{}*+-/=<>≤≥Ω$#∀∃:|",∈'>,
<.,'.'>,<h,h>,<er,'er'>,<bℓ,' '>} .

iv. Next there must follow a set of ordered pairs serving to define the action table of the lexical processor. Each of these pairs has the form

$$<state, <aent_1,aent_2,\ldots,aent_k>>$$.

Here, *state* is a state of the lexical scanner, while each *aent* is an action table entry, having one of the allowed forms described above. The number of *aent* terms shown in the sequence displayed above must equal the number of character types declared; the j-th *aent* term will be consulted when a character of the j-th type is encountered and the lexical scanner is in the specified *state*.

In the case of our hypothetical FORTRAN-like language, we might employ the following lexical states (which, as have already bee noted, correspond to the various states of uncertainty which could arise as we progressively scanned a token from left to right):

nxt - a new token is just starting;
nm - a name is being scanned;
irh - an integer, a real number, or a hollerith constant is being scanned, but we are still not sure which;
ir - having encountered a blank in a string of digits, we are sure that an integer or a real is being scanned;
dip - having encountered a period, we are scanning either a real or an integer followed by a period delimited operator;
r - definitely scanning a real number;
pd - definitely scanning a period-delimited operator.

In the case under consideration, our action table would be described as follows, recalling that corresponding character types are <a,h,'l',+,.,bl,er>);

```
(3)  {<nxt,<<go,nm>,<go,nm>,<go,irh>,<do,spend,end>,
                    <go,pd>, skip, <do,erend,end>> ,
      <nm,<cont,cont,cont,end,end,end,end>>,
      <irh,<end,<do,holcon,end>,cont,end,<go,dip>,<go,ir>,end>>,
      <ir,<end,end,cont,end,<go,dip>, skip,end>>,
      <dip,<<do,back,end>,<do,back,end>,<go,r>,end,end,<go,r>,end>>,
      <r,<end,end,cont,end,end,skip,end>>,
      <pd,<cont,cont,cont,end,end,end,end>>}  .
```

v. Next must follow text defining every auxiliary process mentioned in the state table. This text is supplied as a set of ordered pairs, each having the form

<rname,text> ,

where *rname* is the process name, and *text* is its body. This collection of pairs is converted to a complete body of code having the form

$rname_1$: $text_1$ return;

$rname_2$: $text_2$ return;

. . .

together with a calculated go-to statement which, supplied with identifier, invokes an appropriate auxiliary process.

The auxiliary routines may refer to the lexical analyzer's 'beginning of present token' pointer *tokbegin*, to its 'current symbol' pointer *curpointer*; to *token*, the token being formed; and to the scar state *state*. Note that *nextoken* returns *state*, and *token* to the main program upon encountering an *end* command.

The value of *state* when the lexical action *end* is executed therefore defines the lexical type of *token* to the main program. In certain cases where an *end* is to be executed forthwith, our auxiliary routines may therefore set *state* to values not appearing in the action table. This is merely to inform the main program that some very special situation, such as an end-of-record, has been detected. When *nextoken* is called again, *state* will always be set to *nxt*. Note as an example of this that in the package of auxiliary procedures which follows, the state 'er' indicates end of current record reached, and 'ef' indicates end-of-file.

Considering our hypothetical FORTRAN-like language once more, and noting the occurrences of auxiliary process names in the action table description given above, we would supply the following auxiliary processes.

```
(4)  {<spend,'token=cstring(curpointer); curpointer=curpointer+1;'>,
      <erend,'if cstring(curpointer+1) eq er then
       curpointer = curpointer+2; state = "ef"; token=er+er;
       else state = "er"; end if;'>,
      <back, 'curpointer = curpointer-1;'>,
      <holcon, 'n=dec token; curpointer=curpointer+1;
       if 0 ≤ ∃j < n|cstring(curpointer+j). eq er then
          token = cstring(curpointer:j); curpointer=curpointer+j;
     else token = cstring(curpointer:n); curpointer=curpointer+n;
       end if;'>}
```

Note that (1), (2), (3), and (4) together constitute a complete description of a FORTRAN-like lexical scan. Of course, from another point of view, the text of *nextoken* and of the associated routine *setup* must also be reckoned as part of this description.

After all these preliminaries, we shall now give the SETL code for the lexical *setup* routine. This code has, in a miniature way, many of the features associated with larger compilers. It is in fact a compiler of sorts, transforming tables like those shown above into tables directly interpretable by the rather simple *nextoken* algorithm. Typically enough for programs of this kind, much of *setup* is concerned with verification of the correctness of the data presented to it, and with the printing of diagnostics where required. Aside from this and from some rather straightforward transfo mation of table form, the principal responsibility of *setup* is to build up text for the routine *rpak*. The assumed form for the *rpak* text is as follows.

```
        define rpak(numrout);
        go to {<1,rout1>, <2,rout2>, ...} (numrout);
rout1:  text1  return;
rout2:  text2  return;
        ...
        end rpak;
```

Here, $rout_j$ is the j-th auxiliary procedure name supplied by the programmer, and $text_j$ is the text defining this procedure.

The detailed *setup* code is as follows:

```
/* we begin with some simple auxiliary macros */
macro readcheck(x);  /* check that read ok */
if x eq Ω then print 'run terminated by illformed input';
exit;;  endm readcheck;
macro setype(c); /* adds additional type specification for
     character */
typef{c} = typef{c} with type;  return;  endm setype;
```

```
macro e;  /* error procedure */  errors=errors+1; endm e;

  finef htℓ x; return x(2); end htℓ;
definef pair x; return if type x ne tupl then f
     else (#x) eq 2;  end pair;
define  setup(typ,table,rpak,cstring); /* main setup routine*/
nxt = 'nxt'; errors=0; read allch; readcheck(allch);
allc = {allch(n), 1 ≤ n ≤ #allch} with er;
read ctypes; readcheck(ctypes); typef = nℓ;
(∀tup ∈ ctypes) <type,cstring> = tup;
     if cstring = 'er' then setype(er);
     else (1 ≤ ∀j ≤ #cstring) setype(cstring(j));;
     end if;
end ∀tup;
read seqtypes;  readcheck(seqtypes);
types = {t, t(n) ∈ seqtypes}; types2 = htℓ[typef];
/* check that types  and types2 agree */
if types - types2 is ers ne nℓ then
     print 'types specified but not used are:', ers; e;;
if types2 - types is ers ne nℓ then
     print 'unspecified types are used; these are:', ers; e;;
/* check that all characters have unique type specified */
if {c ∈ allc | typef{c} eq nℓ} is ers ne nℓ then
     print 'type unspecified for following characters:', ers; e;;
if {c ∈ allc | (#typef{c}) gt 1} is ers ne nℓ then
     print 'type multiply specified for following characters:',
       ers; e;;
typ = typef;  read rawtable;  readcheck(rawtable);
     statesused = hd[rawtable];
/* check that 'nxt' belongs to statesused, and that there are
     no repetitions */
if n('nxt' ∈ statesused)then
     print 'required state 'nxt' omitted from table'; e;;
if {st ∈ statesused | (#rawtable{st}) gt 1} is ers ne nℓ then
     print 'multiply defined states:', ers; e;
/* force to single valued function */
     (∀x ∈ ers) rawtable(x) = ∋rawtable{x};;
  | if;
```

```
 /* check that right number of terms in all sequences */
if{st∈statesused|(#rawtable(x)) ne #seqtypes} is ers ne nl then
   print 'states defined with wrong number of type entries:',ers,e,,
/* convert to two dimensional table */
table = {<state,   stp(j),   rawtable(state)(j)>,
        state ∈ statesused,stp(j) ∈  seqtype};
/* check that all non-tuple entries are either end, skip, or cont*/
if{<x,y,table(x,y)>, x ∈ statesused, y ∈ types |
   type table(x,y) is tent ne tupl and n tent ∈  {'end','skip','cont'}
   or(type tent eq tupl and (n (hd table(x,y))∈{'go','do'}}
   or ((hd tent) eq 'go' and (#tent) ne 2)))}is ers ne nl
   then print 'illegal entries in following positions of table:',
    ers;  e;;
/* check that all go-to entries are well-formed */
if {<x,y, table(x,y)>, x ∈ statesused, y ∈ types |
   (hd table(x,y)) eq 'go' and(type table(x,y)) eq tuple and
   n htl table(x,y) ∈ statesused}) is ers ne nl then
print 'illformed go-to entries in following positions of table:',ers;e
/* now prepare to check wellformedness of all call-type entries*/
read routset; readcheck(routset);  routs = hd[routset];
   routscalled = nl;
/* check that all routines are uniquely defined */
if {rt ∈ routs|(#routset{rt}) ne 1} is ers ne nl then
   print 'illdefined or multiply defined routines:', ers; e;;
/* the routine 'callok' used in the next statement is given below.
   it builds up the set 'routscalled' */
if {<x,y,table(x,y)>, x ∈ statesused, y ∈ types |
   pair table(x,y) and (hd table(x,y)) eq 'do'
   and n callok table(x,y)} is ers ne nl
then print 'illegal call-type entries in following positions:',ers;e;
/* check that all routines called are defined */
if {rt ∈ routscalled | n rt ∈ routs} is ers ne nl then
   print 'routines used but not defined:', ers;    e;;
/* give warning diagnostic on superfluous definitions */
if {rt ∈ routs|n rt ∈ routscalled} is ers ne nl then
   print 'warning *-*-*- routines defined but not used:', ers;;
/* number routines */
rnums = nl; (∀r ∈ routs) rnums(r) = #rnums+1;;
```

```
/* set up rpak */
  ktext = 'define rpak(numrout);'+
   'go to {' + [+: rout ∈ routs] ('<'+ dec rnums(rout)+ ',' +rout+'>'
        +if rnums(rout) ne #rnums then ',' else'} (numrout);')
        +[+: rout ∈ routs](rout+':'+ routset(rout)+'return;')
        + 'end rpak;';
/* now replace rout names in action table by corresponding index
   in rpak */
(∀x ∈ statesused, y ∈ types | (hd table(x,y)) eq 'do')

   (∀op(j) ∈ table(x,y))
        if op eq 'do' then
             table(x,y)(j+1) = rnums(table(x,y)(j+1));
        end if;
    end ∀op;
end ∀x;
/* now rpak has been constructed, typef supplied and table constructed*/
return;
end setup;
/* here follows the auxiliary routine callok, used above.
   this routine checks for illformed 'go'- and 'call'-type
   entries in the lexical scan action table, and builds up
   the set routscalled */
definef callok entry;  ok=t;
(∀word(n) ∈ entry)            flow
            ( entry(n-1) eq 'do')?
          putin,                ( entry(n-1) eq 'go')?
                  ( word ∈ statsused)?        shortkeyword?
               cont,              notok,    cont,      longkeyword?
                                                  tooshort?    er,
                                                 er,    cont;

putin: word in routscalled;
shortkeyword := word ∈ {'end','skip','cont'};
longkeyword :=  word ∈ {'go', 'call'};
tooshort := (#entry) eq n;
  it:  continue;
```

```
er:  return f;
notok:  ok = f;                    end flow;
end ∀word;  return ok;
end callok;
```

3. Miscellaneous combinatorial algorithms.

In the next few paragraphs, we will write out various algorithms of a rather mathematical flavor, diversely related to combinatorial stiuations of theoretical interest. These algorithms are intended to demonstrate the ease with which SETL adapts itself to a variety of combinatorial structures and situations. Our first example is simple but famous: Cantor's 'diagonalizer', which, given a set s and a multivalued map f: s → s, produces a set which is not of the form f{s}. It is

```
diagset = {x ∈ s | n x ∈ f{x}};
```

the reader may supply the proof.

Next we present some useful "closure" algorithms. If *as* and *bs* are subsets of a set *s*, and *f* is a (possibly multivalued) map on *s*, the following sets are often of interest. The set *close(f,as)* consists of all points obtained by repeated applications of *f* to *as*; the set *closure(f,as,bs)*, consists of all points obtained by repeated application of *f* to *as*, taking only images in *bs*. The corresponding SETL algorithms are as follows.

```
definef close(f,as);
im = f[as]; n = 0;
(while n lt #im) n = #im; im= im+f[im];;
return im;
end close;
definef closure(f,as,bs);
im = f[as] * bs;
  fp={g∈f|(g(1) ∈ bs and g(2)∈bs)};
n = 0;
(while n lt #im) n = #im; im=im + fp[im];;
return im;  end closure;
```

Given a map f, the following algorithm returns f' such that (x) = close(f,{x}):

```
definef closef(f,set);
fp = f;
(∀x ∈ set) n = 0;
     (while n lt #fp{x})
          n = #fp{x}; fp{x}= fp{x}+ fp[fp{x}];
     end while;
end ∀x;
return fp;
end closef;
```

It should be noted that the above algorithms are deliberately given in simple short forms, even though considerably more efficient though more complex forms are available for some of them.

Next we give an algorithm related to the problem of maximum network flow, which turns out to be central to an interesting group of combinatorial algorithms, some of which at first sight seem to have no contact with this problem. By a *network* we mean a collection N of points p, such that for each pair <p,q> of distinct points in N a non-negative *capacity* c(p,q) is defined. We may think of c(p,q) as representing the maximum 'fluid carrying capacity' of a 'pipe' connecting p and q, and oriented from p to q. A 'pipe' from p to q is 'absent' whenever c(p,q) = 0. Note that the values c(p,q) and c(q,p) are independent; in effect, therefore, networks are *oriented*. A flow in the network is a function f(p,q) which assigns to each pair of distinct points a value satisfying $0 \leq f(p,q) \leq c(p,q)$. If merely $f(p,q) \geq 0$, but the condition $c(p,q) \geq f(p,q)$ can be violated for some p,q, then we call f an *overflow*. The net f-*outflow* from a point p is the sum over all $q \neq p$ of f(p,q) - f(q,p); the f-*inflow* to p is the negative of this quantity. Given two distinct points x,y of N, we say that f is a flow *from x to y* if the net f-outflow from each point other than x and y is zero and the net f-outflow from x is nonnegative. We define the motion of an *overflow* from x to y similarly. It is hard to see that in a flow (or overflow) from x to y the net inflow to y must equal the net f-outflow from x. This common

value is called the *value* (or *transport value*) of the flow f. The *maximum flow* problem is the problem of finding a flow of maximum transport value from x to y in a network, given the capacities c(p,q).

We attack this problem as follows. Let f be a flow from x to y. If both f(p,q) and f(q,p) are positive, let m be their minimum and put $\bar{f}(p,q) = f(p,q) - m$, $\bar{f}(q,p) = f(q,p) - m$. It is clear that f is still a flow from x to y; we call $\bar{f}$ the reduction of f, and observe that $\bar{f}$ and f have the same transport value. Next, let $p_0, p_1, \ldots, p_n$ be a sequence of points of N, the first element p_0 of the sequence being identical with x, the last p_n being identical with y. We call such a sequence an x,y-path. Designating such a path by π, we put $f_\pi(p,q) = \sum_{j=1}^{n-1} f_{(p_j,p_{j+1})}(p,q)$, where

(1) $f_{(u,v)}(p,q) = 1$ if $\langle u,v\rangle = \langle p,q\rangle$, $f_{(u,v)}(p,q) = 0$ otherwise.

It is clear that f_π is an overflow from x to y, and that its value is 1. Let a flow f from x to y be given, and let its value be V. If there exists a positive constant γ such that the reduction g of the sum $f + \gamma f_\pi$ is a flow (necessarily from x to y), then it is clear that the value of g is $V + \gamma$; g is therefore a flow from x to y having value larger than that of f. The condition that such a path π and number $\gamma > 0$ should exist may clearly be formulated as follows: there must exist an x,y-path $p_0, p_1, \ldots, p_n$ such that for each $0 \leq i < n$ we have either $f(p_{i+1},p_i) > 0$ or $f(p_i,p_{i+1}) < c(p_i,p_{i+1})$. Call such a path an *f-augmenting* path from x to y; we restate the observation just made, as follows: given an f-augmenting x,y-path we can at once produce a flow from x to y having value larger than that of f. Conversely, suppose that no augmenting path from x to y exists. Then let X be the set of all points which can be reached along an f-augmenting path starting at x. Let $\bar{X}$ be the complement of X. Clearly $y \in \bar{X}$, and clearly $f(p,q) = c(p,q)$ and $f(q,p) = 0$ if $p \in X$ and $q \in \bar{X}$. It is easy to see from this that the value V of f is equal to the sum

(2) $$\sum_{p\in X, q\in \bar{X}} c(p,q) .$$

Since, as easily established, no flow from x to y can have a value rger than (2), it follows that f is a flow of maximum value.

The above remarks prove the following theorem.

Max-Flow Min-Cut Theorem. Given a network defined by a set of capacities c(p,q), and given two points x,y in the network, the maximum value which any flow f from x to y can have is at the same time the minimum value of the expression (2), where in (2) X ranges over all sets containing x but not y.

Our argument also gives us the following algorithm for constructing a flow of maximum value, at least in case the capacities c(p,q) are all integral. We start with the 'trivial' flow f, for which f(p,q) is identically zero. If there exits an f-augmenting x,y-path π, we replace f by the reduction g of $f + \gamma f_{\pi}$, taking γ to be as large as possible subject to the requirement that g be a flow. Note that since all the capacities c(p,q) are assumed to be integral, γ will be an integer, and g will have a value exceeding that of f by at least 1. Hence, iterating our construction a finite number of times, we will eventually obtain a flow from x to y having maximum value.

We shall now give a SETL code representing the procedure just outlined. The following remarks will aid the reader in following this code. The main routine which appears is *maxflow*, which takes as arguments a set of pairs called *graph* and an integer-valued capacity function c(p,q) defined for $\langle p,q \rangle \in$ graph and $p \neq q$. The set *nodes* is the set of all points appearing in a pair belonging to *graph*; since in practical situations c(p,q) will be zero except for a relatively small set of pairs $\langle p,q \rangle$, we prefer to work from *graph* rather than from *nodes*.

The routine *path*, given two points x,y and a flow f, constructs and returns an f-augmenting path from x to y if possible. If this is impossible, *path* returns the value Ω.

Note also that the binary function a orm b returns *a* unless *a* is Ω, in which case it returns *b*.

```
definef r(e); /* reversed edge */ return <e(2),e(1)>; end r;
definef maxflow(x,y,graph,c); /* main routine */
gr = graph + r[graph];
nodes = {e(1), e ∈ gr};
f = {<e,0>, e ∈ gr};
(while path(x,y) is p ne Ω)
auxflowv = [min: e ∈ p] cap(e,f,c);
   (∀e ∈ p)
        f(e) = f(e) + auxflowv;
        redund = f(e) min f(r(e));
        f(e) = f(e) - redund;  f(r(e))=f(r(e))-redund;
   end ∀e;
end while;
return f;
end maxflow;

definef cap(e,f,c);
return f(r(e)) max (c(e)orm 0 - f(e));
end cap;

definef path(x,y); /* constructs f-augmenting path if possible */
/* we assume in this routine  that gr, f, and c are global */
new ={y}; set = new;
next = nl; /* next will point along the nodes of a path */
(while new ne nl doing new = newer;)
   newer = nl;
   (∀v ∈ new)
        prior={u∈gr{v}|u n ∈ set and cap(<u,v>,f,c) gt 0};
        (∀u ∈ prior)<u,v> in next;
             if u eq x then go to done;;
             u in set; u in newer;
        end ∀u;
   end ∀v;
end while;
```

```
    /* loop fallout means path is impossible */ return Ω;
dc  pth = nℓ; pt = x; /* now loop to build up path */
    (while next(pt) ne Ω doing pt = next(pt);)
        <pt, next(pt)> in pth;
    end while;
    return pth;
    end path;
```

Next we consider the so-called combinatorial "matching" or "marriage" problem, and an algorithm solving it. The problem is this: given a multivalued map from a set s to a disjoint set t, when can we find a one-to-one map g defined on s such that $g(x) \in h\{x\}$? The necessary and sufficient condition is that #h[t] ge #t for each subset t of s. More generally, we may ask: what is the maximum number of points in a subset $\bar{s}$ of s on which there exists a one-to-one g such that $g(x) \in h\{x\}$? Answer: $\#\bar{s}$ equals the total number of points in s, minus the maximum m of $\#\hat{s} - \#h[\hat{s}]$, $\hat{s}$ ranging over all subsets of s. This is the so-called matching theorem. To prove it, note that a single valued g with $g(x) \in h\{x\}$ must fail to be defined on at least $\#\hat{s} - \#h[\hat{s}]$ points of $\hat{s}$, and hence on at least m points of s. To prove conversely that g may be defined on all but m points of s, we make use of the max-flow min-cut theorem, in the following way. Introduce two points x and y distinct from all the points p of the union of s and t. Now define capacities as follows: c(x,p) = 1, for $p \in s$; c(p,q) = 1 if $p \in s$ and $q \in h\{p\}$; c(q,y) = 1 if $q \in t$; c(p,q) = 0 in all other cases. Using these capacities, let f be a flow from x to y having maximum value; by what has been proved in preceding pages, we may take all the values of f to be integral. Clearly then, each value of f is either 0 or 1. Put g(p) = q if f(p,q) = 1; since the total 'outflow capacity' from q is c(q,y), i.e., is 1, the function g is one-to-one. Our definition of c makes it plain that $g(p) \in h\{p\}$ for all p. The value of the x,y-flow f is clearly not more than the number of points in the domain of the 1-1 function g. By the max-flow min-cut theorem, this value equals minimum sum $\sum_{p\in X, q\in \bar{X}} c(p,q)$, where $\bar{X}$ is the complement of X and where the set X includes the point x but not the point y.

This minimum is clearly also the minimum of

(3) $$\#(s-u) + \#(h[u]-v) + \#v \ ,$$

where u ranges over all subsets of s and v over all subsets of t. But for fixed u the minimum of (3) as v varies is clearly

(4) $$\#(s-u) + \#h[u] \ .$$

Thus at least $\#s - \max_u (\#u - \#h[u]) = \#s = m$ points belong to the domain of g, proving the matching theorem.

The proof just given clearly reduces the construction of the 1-1 function g to a maximum flow construction and gives us the algorithm for 'maximum match' expressed in the following SETL code:

```
definef maxmatch(h);
/* h is a possibly multi-valued function.
   we seek a 1-1 g such that g(x) ∈ h{x} */
x = newat;  y = newat;
graph = h + {<x, z(1)>, z ∈ h} + {<z(2),y>, z ∈ h};
c ={< e,1>, e ∈ graph};
mf = maxflow(x,y,graph,c);
return {p ∈ h | mf(p) ne 0};
end maxmatch;
```

This construction has a wide variety of interesting extensions. Suppose, to give just one example, that nm(x) is a numerically valued function defined on the domain s of h, and that we are required to construct a multi-valued function g such that g{x} is always a subset of h{x}, such that g{x} always contains nm(x) elements, and such that all the sets g{x} are disjoint. (For nm(x) ≡ 1, this reduces to the case that has been considered.) The necessary condition is clearly $\#h[t] \geq \sum_{x \in t} nm(x)$ for all subsets t of s. We may easily see that this condition is also sufficient as follows: replicate each point x ∈ s, nm(x) times; put $h\bar{x} = h\{x\}$ for each replica $\bar{x}$ of x. Solve the matching problem with nm ≡ 1 for h on this larger domain. This clearly gives a solution to our generalized matching problem. In SETL, this construction may be written as

```
definef genmatch(f,nm); nf = nl;
  plicas = {<x,x,k>, x ∈ hd[f], 1 ≤ k ≤ nm(x)};
(∀x ∈ hd[f], y ∈ replicas{x}) nf{y} = f{x};;
return  maxmatch(nf) c replicas; /* the binary function f c g
   'compounds' the maps f and g */
end genmatch;
```

Here we have employed the generally useful functional composition operation defined as follows.

```
define f c g;  return {<x,y>, x ∈ hd[g], y ∈ f[g{x}]};  end c;
```

We continue our sequence of combinatorial algorithms by discussing the so-called *Menger curve theorem*, which we state in a form applying to *unordered graphs*. An unordered graph is a set of *nodes*, together with a symmetric relationship between pairs of nodes, which tells us when a pair of nodes is connected by an edge, i.e., are *neighboring*. We may also think of this as a multi-valued function *naybs*, defined on s, which gives us the set of neighbors of each node, and which has the property that x ∈ naybs{y} implies y ∈ naybs{x}. If this property does not hold, we call our graph *ordered*.

A pair <y,x> such that x ∈ naybs{y} is called an *edge* of the graph. Given two sets a and b in a graph, we say that they can be *connected* if there exists a sequence of nodes, the first in a, the last in b, the second member y of every successive pair x,y of nodes belonging to naybs{x}; such a sequence is called a *path* from a to b. We say that a and b are n-*connected* in the graph if, whenever n-1 nodes are removed from the graph, the parts of a and b which remain are still connected. Menger's theorem asserts that if a and b are n-connected in the graph, there exist at least n paths connecting a and b and having no points in common.

Instead of proving this result directly, we will find it useful to relate it to a similar result concerning ordered graphs and disjoint paths. Given two sets a and b in an ordered graph, we say that a is n-*path-connected* to b if, however n-1 edges removed from the graph, there still exists a path from a to b.

A result on ordered graphs related closely to Mengers' theorem on unordered graphs states that if a is n-path-connected to b then there exist at least n paths from a to b having no edges in common.

This last assertion may be derived in the following way from the max-flow min-cut theorem. Use the graph g containing a and b to define a network N as follows: The points of N are the points of g, plus two additional points x and y. Capacities are defined for the edges of N as follows: $c(p,q) = 1$ if $q \in naybs\{p\}$; $c(p,q) = 0$ otherwise, $c(x,p) = L$ for $p \in a$ and $c(q,y) = L$ for $q \in b$, where L is a large integer (how large L must be will appear below). Then find a flow f from x to y having maximum value V_f. By the min-cut max-flow theorem, V is equal to the minimum of

$$m_\sigma = L * (\#(a - \sigma) + \#b * \sigma) + \#(naybs[\sigma] - \sigma) ,$$

σ ranging over all subsets of s. For L large this minimum is clearly attained for a σ which includes a and is disjoint from b. Now, by removing from σ all edges going from σ to a point outside σ no path from a to b remains. Hence since a is n-path connected to b in the graph g, it follows that $m_\sigma \geq n$, so that there exists a flow f in N having value V_f at least n.

Call a path π_1 from x to y in the network N f-*dominated* if $f(p,q) = 1$ whenever p,q are successive points of π_1. Given that $V_f \geq 1$, there must exist an f-dominated path from x to y. Indeed, suppose the contrary, and let X denote the set of all points which can be reached by proceeding from x along an f-dominated path. Then y belongs to the complement $\bar{X}$ of X,and no edge $\langle p,q \rangle$ of the graph g with initial point $p \in X$ and terminal point $q \in \bar{X}$ satisfies $f(p,q) > 0$. Using the general formula

$$|V_f| \leq \sum_{p \in X, q \in \bar{X}} f(p,q) ;$$

we deduce at once that V_f must be zero, a contradiction which proves the existence of an f-dominated path π_1. Then letting f_{π_1} be defined as in formula (1) above, and putting $f' = f - f_{\pi_1}$

we see that if $V_f = n$ the above construction can be repeated n times, to give n distinct f-dominated paths $\pi_1,\ldots,\pi_n$ from x to y, no two of these paths having an edge in common. The paths $\pi_1,\ldots,\pi_n$ clearly include n arc-disjoint paths connecting the set a to the set b.

The construction just described is summarized in the following SETL algorithm, which, given an oriented graph g defined by a set *nodes* and a function *naybs*, and given two subsets a,b of the set *nodes*, finds the largest possible number of edge-disjoint paths between a and b. Two principal auxiliary routines are used. The first, *fpath*, constructs an f-dominated path between two designated points; the second, *dimin*, performs the operation $f = f - f_\pi$, π being some specified path.

```
definef disjpaths(nodes,naybs,a,b);
x = newat; y = newat;
net = naybs + {<x,z>, z ∈ a} + {<z,y>, z ∈ b};
large = #nodes;
/* it is easily seen that this quantity is sufficiently large
   to play the role of the quantity L in the preceding discussion*/
/* r is the 'edge-reversing' function used also in the 'maxflow'
   algorithm */
rnet = r[net];
c = {<e,1>, e ∈ naybs} + {<<x,z>, large>, z ∈ a}
        + {<<z,y>, large>, z ∈ b>};
mf = maxflow(x,y,net,c);
pset = nl;
(while fpath(x,y,mf) is path ne Ω doing dimin(mf,path);)
   path(2:#path-2) in pset;
    /* thus dropping the auxiliary links to x and y */
end while;
return pset;
end disjpaths;
define  dimin(mf,path);
(∀x ∈ path) mf(x) = mf(x)-1;;
   urn;
end dimin;
```

```
definef fpath(x,y,mf); /* rnet is assumed to be global */
new = {y}; set = new;
next = nl; /* next will point along the nodes of a path */
(while new ne nl doing new = newer;)
   newer = nl;
   (∀v ∈ new)
        prior={u∈ rnet{v}|n u ∈ set and mf(<u,v>) gt 0};
        (∀u ∈ prior)<u,v>  in next;
             if u eq x then go to done;;
             u in set; u in newer;
        end ∀u;
   end ∀v;
end while;
/* loop fallout means path is impossible */ return Ω;
done: pth = nl; pt = x;
     (while next(pt) ne Ω doing pt = next(pt);)
          <pt, next(pt)> in pth;
     end while;
     return pth;
     end fpath;
```

Now we return to Menger's theorem in its original form, i.e., to the problem of constructing n disjoint paths between two sets a and b n-connected to each other in an unordered graph g. We proceed by introducing an ordered graph $\bar{g}$ as follows. For each node p of g, $\bar{g}$ contains two nodes, which we designate as p^{in} and p^{out} respectively. The edges of $\bar{g}$ consist of the pairs $<p^{in},p^{out}>$, together with all pairs $<q^{out},p^{in}>$, where <q,p> is an edge of g. Note then that if a path $p_1,p_2,\ldots,p_n$ in g is given, it defines a path $p_1^{in},p_1^{out},p_2^{in},p_2^{out},\ldots,p_n^{in},p_n^{out}$ in $\bar{g}$. Conversely, since in $\bar{g}$ there exists an edge going from p^{in} to q^{out} if and only if p = q, it follows that every path $\bar{\pi}$ connecting p_1^{in} to p_n^{out}, must have the form $p_1^{in}, p_1^{out}, p_2^{in}, p_2^{out}, \ldots, p_n^{in}, p_n^{out}$, i.e., that $\bar{\pi}$ must correspond to a path π connecting p_1 to p_n in α. It is clear that if $\bar{\pi}_1$ and $\bar{\pi}_2$ are two such paths in $\bar{g}$, and if

π_1 and π_2 are the two paths in g which correspond to $\bar{\pi}_1$ and $\bar{\pi}_2$ spectively, then π_1 and π_2 are point-disjoint if and only if $\bar{\pi}_1$ and $\bar{\pi}_2$ have no edges in common. Thus the problem of constructing n disjoint paths between two sets a and b in g is reduced to that of constructing n edge-disjoint paths between two corresponding sets in $\bar{g}$.

The following SETL algorithm makes use of the abstract construction described in thepreceding paragraph.

```
definef menger(nodes,naybs,a,b);
nodesbar = {<n,0>, n ∈ nodes} + {<n,1>, n ∈ nodes};
naybsbar = {<<n,1>,<n,0> , n ∈ nodes}
          + {<<n,0>,<m,1>>, n ∈ nodes, m ∈ naybs{n}};
p = disjpaths(nodesbar,naybsbar,{<n,1>, n ∈ a},{<m,0>, m ∈ b});
return altpts[p];
end menger;

definef altpts(pibar);
return [+: x(n) ∈ pibar | n//2 eq 0] <hd x>;
end altpts;
```

The preceding algorithms are intended to illustrate the fact that set-theoretic constructions of the type conventional in many mathematical discussions can very readily be written out in SETL. Another interesting example of this point is furnished by *Dilworth's Theorem*, which states that if s is a partially ordered set, i.e., a set on which a transitive relationship xRy is defined, then the minimum number n of *completely ordered subsets* or *chains* by which s can be covered is equal to the maximum number m of elements which can be found in s, no two of which are related by the relation R. Here we call a subset t of s *completely ordered*, or a *chain*, if given any x and y in t, we have either xRy or yRx. We suppose for simplicity that if xRy and yRx, then x = y. It is plain that $n \geq m$, thus only $m \geq n$ need be proved. The following observation of D. R. Fulkerson, pointed out to me by A. Hoffman, reduces Dilworth's theorem to the matching theorem. Write

x if yRx and $y \neq x$. Let f map each element y of s onto the

set of all y such that yR'x, and use the matching theorem to find a maximally large 1-1 mapping g subordinate to this mapping, i.e a 1-1 mapping such that g(x)Rx defined for as many elements as possible. By the matching theorem, the number of elements for which g is undefined is $\bar{m}$ = max(#t - #f[t]), where f[t] is the set of all elements such that yR'x for some $x \in t$. Given a subset t of x for which the maximum is attained, form t' = t - f[t]. It is clear that we never have xR'y for $x,y \in t'$; thus t' consists only of unrelated elements, so $m \geq \bar{m}$. Suppose now that we take the collection of all those elements u such that $u \neq g(v)$ for any $v \in s$, and then form $p = \langle u, g(u), g^2(u), \ldots, g^k(u)\rangle$, until finally $g^{k+1}(u)$ is undefined. Then it is clear that the p are a family of chains covering s, and that their number is at most $\bar{m}$; so that $n \leq \bar{m} \leq m$.

This construction may be represented in SETL as follows:

```
definef dilworth(set,reln);
/* we assume that reln is a function re urning the values t, f */
h = {<x,y>, x ∈ set, y ∈ set | reln(x,y) a x ne y};
g = maxmatch(h); bots = set - g[set]; dil = nl;
(∀x ∈ bots) y = x; tup = <y>;
   (while g(y) ne Ω doing y = g(y);)
        tup = tup + <g(y)>;
   end while;
   tup in dil;
end ∀x;
return dil;
end dilworth;
```

Dilworth's theorem has an interesting connection with the problem of representing, in compact form, a *binary relationship* over a given set s. Such a relationship is a 0,1-valued function r(x,y), defined for all $x \in s$, $y \in s$. A relationship of this kind can of course be represented (at machine level) by a table of bits, n × n in size, where n is the number of elements of s. Various other possibilities, which may in certain cases be more efficie exist. For example, we may store, with each x, the set of all y

for which r(x,y) = 1, representing this set as a list. Here we shall consider another way of representing r(x,y), which is advantageous in certain cases. Our idea is as follows. First associate with each x the set

$$U_x = \{y \in s \mid r(x,y) \text{ eq } 1\}.$$

If the family of sets U_x can be arranged in a single, steadily increasing family $U_{x_1} \subseteq U_{x_2} \subseteq U_{x_3} \subseteq \ldots \subseteq U_{x_n}$ of sets, then we can assign, to each x in s, the index of the set U_x in this sequence, thereby defining a function f(x). If, for each y, we let g(y) be the index of the smallest set U_x to which y belongs, then plainly r(x,y) eq 1 is equivalent to $g(y) \leq f(x)$. Hence r(x,y) can be represented by a table of 2n values, rather than by a table of n^2 bits.

In general, we cannot expect that the sets U_x will form a single, steadily increasing chain. Neverhteless, the family of all sets U_x is partially ordered, and we can use Dilworth's theorem to break the collection of all the sets U_x into a minimum number of separate steadily increasing chains of sets. We may then let f(x) be the index of U_x in the chain c(x) to which U_x belongs, and, for each $y \in s$ and each chain c, let g(y,c) be the index of the smallest set, in the chain c, which contains y. It then follows that r(x,y) eq 1 is equivalent to $g(y,c(x)) \leq f(x)$. If k chains are needed to accommodate all the sets U_x, this method allows us to represent the relation r(x,y) by a table of (k+2)n, rather than by a table of n^2, values.

This compaction procedure, quite useful in certain cases, may be written as follows in SETL.

```
definef compactreln(set,relation);
relpairs = {<x,y>, x∈set, y∈set|
              (∀ z ∈ set | relation(y,z) imp relation(x,z))};
chains = dilworth(set,reln);
f = nl; g = nl; c = nl;
(∀ch ∈ chains,  x(j) ∈ ch) f(x) = j;  c(x) = ch;;
   ∈ set, ch ∈ chains) g(y,ch) =
  if  ∃x(j) ∈ ch|relation (x,y)  then j else #ch+1;;
return <f,g,c>;
```

```
end compactreln;
/* for the present application, the function reln called by
   dilworth  is defined as follows */

definef reln(x,y); /* relpairs  is assumed global */
return <x,y> ∈ relpairs;  end reln;
```

Next we give a set of combinatorial routines of a rather different character. These are all combinatorial *generators*, i.e., algorithms which generate all the members of a certain interesting class of combinatorial objects. The first of these is a program to generate all permutations of n objects, which we give here because of its relation to other of the algorithms to be given in this section, even though most of our discussion of algorithms related to permutations is reserved for later pages.

This program generates all permutations of n. Like most of our other generators, it works by iterative application of a rule which defines the k+1-st object to be generated in terms of the k-th object; where the enumeration of objects is to be in some well-defined, generally lexicographic, order. For permutations, we do in fact use the standard lexicographic order. Then the next permutation after a given s_n is defined by the following rule: increase the last possible element by the smallest possible amount. That is, given s_n , find the last element s_j which is not part of a monotone decreasing "tail", interchange it with the smallest s_k with $k > j$ and $s_k > s_j$, and then place all the elements $s_{j+1},\ldots s_n$ into ascending order. In the program which now follows, a signal is transmitted through "more" when the process restarts.

```
definef perm(n,more);
/* initialize if new */
if n more then more = t;  seq=[+: 1≤k≤n] <k>; return seq;;
/* if sequence is monotone decreasing, there are no more
   permutations. Otherwise find last point of increase*/
if n (n > ∃j ≥ 1 | seq(j) lt seq(j+1)) then
   more = f;  return Ω;
end if;
```

```
/* then find the last seq(k) which exceeds  seq(j) and swap */
  nd = n ≥ ∃k > j | seq(j) lt seq(k);
<seq(j),seq(k)> = <seq(k),seq(j)>;
/* then rearrange all the elements after seq(j+1) into
   increasing order */
(j < ∀k ≤ (n+j+1)/2)
   kk = n - k + j + 1;
   <seq(k),seq(kk)> = <seq(kk),seq(k)>;
end ∀k;
return seq;
end perm;
```

A routine for generating the elements of the set npow(n,s) may be based on the same principle. We arrange the elements of s in a sequence, and always arrange each subset of s in the increasing order of this sequence. The rule for obtaining the next subset after a given one is again: increase the last possible element by the smallest possible amount; and reduce all the elements which follow it by the greatest possible amount. In SETL we have:

```
definef nexnpow(n,s,more);
if n more then
   more = t;  seq = nl;
   (∀x ∈ s) seq(#seq+1) = x;;
   select = {<i,i>,1≤i≤n};
   return seq[tl[select]];
end if;
if select(n) ne #seq then
   select(n) = select(n)+1;
   return seq[tl[select]];
end if;
if n (n > ∃j ≥ 1 | select(j+1) gt select(j)+1) then
   more = f; return Ω;
end if;
(n ≥ ∀k ≥ j) select(k) = select(j)+k-j+1;;
   urn seq[tl[select]];
end nexnpow;
```

Of course, a complete power set generator is easily defined using this routine.

```
definef nexpow(s,more);
if n more then more = t; n = 1; mor = f; return nl;;
subset = nexnpow(n,s,mor); if mor then return subset;;
if n eq #s then
   more = f; return Ω;
else n = n+1; mor = f;
   return nexnpow(n,s,mor);
end if;
end nexpow;
```

Again using the same idea, we may define a routine which generates all the maps of one set into another.

```
definef nexmap(from,to,more);
if n more then
   more = t; fseq = nl;
   (∀x ∈ from) fseq(#fseq+1) = x;;
   tofol = nl; prev = Ω; /* now chain elements of to together in a list *
   (∀x ∈ to) if prev ne Ω then <prev,x> in tofol;
        else first = x;
        end if;
        prev = x;
   end ∀x;
   map = {<x,first>, x ∈ from};
   return map;
end if;
if n (#from ≥ ∃j ≥ 1 | tofol(map(fseq(j))) ne Ω) then
   more = f; return Ω;
end if;
map(fseq(j)) = tofol(map(fseq(j));
(j < ∀k ≤ #from) map(fseq(k)) = first;;
return map;
end nexmap;
```

Next we give a generator of all the partitions of n (i.e. distinct combinations of positive integers with sum n), constructed along

similar lines. The ordering scheme used is as follows. Partitions $n_1,\ldots,n_k$ with $n_1+\ldots+n_k = n$ are kept in monotone decreasing arrangement, i.e., $n_1 \geq n_2 \geq \ldots \geq n_k > 0$, and are generated in lexicographic order.

The next partition after $n_1,\ldots,n_k$ is then obtained by finding the largest j such that n_j can be increased (either j = 1 or the largest $j < k$ such that $n_{j-1} > n_j$), incrementing n_j by 1, and setting all the n's after n_j to 1. Here is the SETL code.

```
definef nexpart(n,more);
if n more then
   more = t;  pseq=[+: 1≤k≤n]<1>;
   return pseq;
end if;
if(#pseq eq 1 then more = f; return Ω;;
if n #pseq > ∃j > 1 | pseq(j-1) gt pseq(j) then j = 1;;

pseq(j) = pseq(j)+1;  tot =([+: j<k≤#pseq]pseq(k))-1;
pseq = pseq(1:j) + ([+: 1≤k≤tot]<1> orm nult);
return pseq;
end nexpart;
```

Next we generate various types of trees using a similar technique. Since trees have a recursive structure, we shall want to code the advance from one tree to another as a recursive process. First consider the problem of generating all *binary trees* with n nodes. We order these trees by the following recursive principle: suppose that all trees of less than n nodes have already been ordered. Then, among trees with exactly n nodes, we put all those with a smaller number of nodes down the left-hand branch from the root ahead of those with a larger number of nodes down the left hand branch from the root. Among those with a given number of nodes down the left-hand branch from the root, we order trees first according to their left-hand subtree, then according to their right-hand subtree. The first tree is that in which no node has a left-hand successor. To advance from a tree to the next tree, we

i. Advance the right-hand subtree to its successor, if possible.

ii. If this is impossible, advance the left-hand subtree, and set the right-hand subtree to the first tree with the same number of nodes.

iii. If this is impossible, transfer one node from right to left, and set both the right and left-hand subtrees equal to the first tree with the appropriate number of nodes.

The following figure shows all three-node binary trees in the order in which they will be generated.

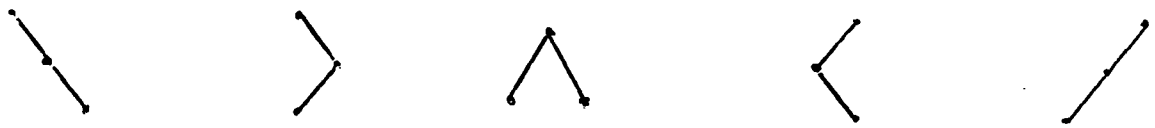


Fig. 3. All binary trees with three nodes.

In SETL, the tree-generation routine is as follows:

```
definef nextree(n,more);
if n more then
   more = t; ℓ = nℓ; r = {<j,j+1>, 1≤j<n};
   top = 1;
   return <ℓ,r,top>;
end if;
advance(top,more);
return if more then<ℓ,r,top> else Ω;
end nextree;

define  advance(top,more);
if r(top) eq Ω then
   if ℓ(top) eq Ω then more = f; return;; /* else */
   advance(ℓ(top),more); return;
else
   advance(r(top),more);
end if;
if more then return;; /* otherwise */
nodesetr = nodesof(r(top)); advance(ℓ(top),more);
if more then r(top) = newtree(nodesetr); return;; /* otherwise */
   nodesetr = nodesetr less r(top);
nodsetℓ =    nodesof(ℓ(top)) with r(top);
ℓ(top) = newtree(nodesetl); r(top) = newtree(nodesetr); more=t; return
end advance;
```

```
definef nodesof(top);
   urn if top eq Ω then nℓ else
   nodesof(ℓ(top)) + nodesof(r(top)) with top;
end nodesof;

definef newtree(set);
if set eq nℓ then return Ω;;
prev from set; first = prev; ℓ(first) = Ω;
(∀x ∈ set) ℓ(x) = Ω;
   r(prev) = x;
   prev = x;
end ∀x;
r(prev) = Ω;
return first;
end newtree;
```

As a final example of this class, we consider the problem of generating all *unordered trees* with n nodes. Such a tree is defined by a set of n nodes, one of which is designated as the head of the tree; and by a multivalued mapping *cesors* of this set into itself, which defines the set of successors of each node. We require that the iterated application of this map to the head should eventually produce every other node; and that, for each node x, the iterated application of this map to x can never produce x. This last condition amounts to requiring that a tree be *cycle free*. We assign a standard *sequence number* to unordered trees recursively, as follows. Suppose that sequence numbers have already been defined for all unordered trees of less than n nodes. Given a tree of n nodes, take its root x and take all the subtrees whose roots are the immediate successors of x. Arrange these successors first by decreasing order of the number of nodes they contain, and, if two contain the same number of nodes, then in decreasing order of their sequence numbers; we call this order the standard ordering of the unordered tree. This associates a sequence S of sequence numbers with each unordered tree T of n nodes. By arranging all such sequences S in lexicographic order, and assigning the position of S in this lexicographic order to T as its sequence number, we define a sequence number for each unordered tree with n nodes.

In this standard sequencing of trees with n nodes, the first is that in which the root has n-1 successors. To advance from a tr T to the next tree, we proceed as follows:

i. Take T in its standard ordering;

ii. Among the subtrees whose roots are immediate descendants of the root R of T, find the last L which it might be possible to advance. This is either the first subtree, or the last subtree which either has fewer nodes than, or the same number of nodes and a lower sequence number than, its immediate predecessor.

iii. If possible, advance L, and make all the nodes belonging to later subtrees immediate successors of R.

iv. If L cannot be advanced, collect all the nodes belonging to later subtrees into a set S. Transfer one node out of S, into L. Redefine L as the first tree with its new number of nodes; make all the nodes remaining in S into immediate successors of R.

v. If the root of L is the last descendant of R, find the last immediate subtree of R prior to L which it might be possible to advance, and proceed with L' as in case iv.

The following figures show all unordered trees of three, four, and five nodes, in the order in which they would be generated by the algorithm just outlined.

Fig. 4. Unordered trees of three nodes.

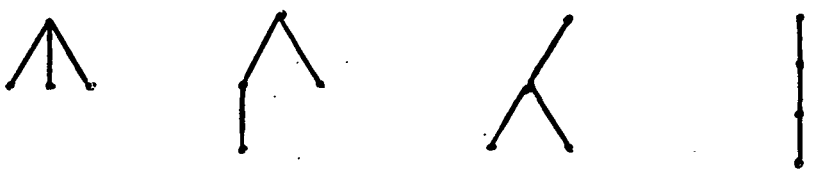


Fig. 5. Unordered trees of four nodes.

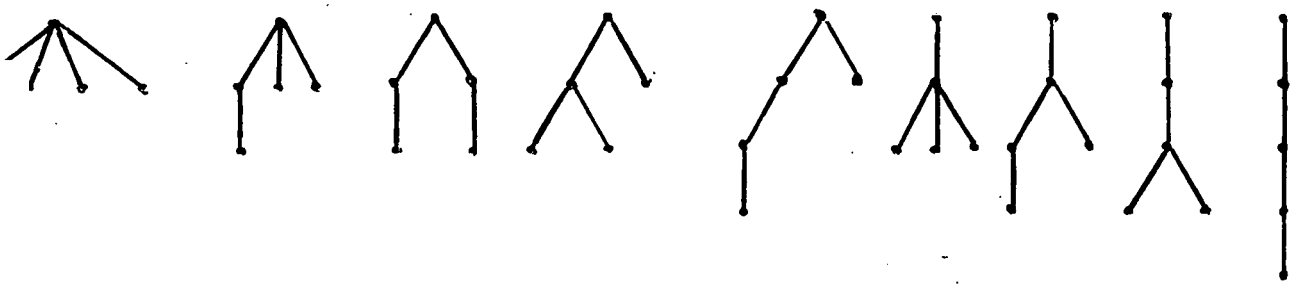


Fig. 6. Unordered trees of five nodes.

The following observations will clarify salient details of the SETL algorithm given below. In place of unordered trees (defined by a set of nodes and a multivalued 'descendant' function desc{n}) we keep ordered 'representatives', in which each node n is mapped into a sequence odesc(n,j). With each node n we also keep the sequence number seqno(n) of the subtree T having n as root, and the number numnodes(n) of nodes in T. This auxiliary information is used during the necessary tree-advance process.

The SETL algorithm is as follows; note the use of the auxiliary function x orm y , which returns if x ne Ω then x else y.

```
definef nexotree(n,more);
if more then go to try; else
/* initialize */
   more = t; top = 1;  odesc = nl;
   numnodes = {<top,n>}; seqno = {<top,1>};
   hang(top,1,{j, 1<j≤n});
/* the subroutine 'hang' adds a set of nodes as immediate successors
   of a given node x, starting at a given position j. Each node hung
   is assigned numnodes = 1; seqno = 1; all prior immediate succes-
   sors of x, beginning at the position j, are removed */
   cesors = nl;
   end if;
<ret:>(1 ≤ ∀node ≤ n) cesors{node} = {x(2), x ∈ odesc{node}};;
   return <top,cesors>;
<try:> advance(top,more);
   more then go to ret; else return Ω;;
end nexotree;
```

```
define advance(top,more);
k = # odesc{top}; if k eq 0 then more = f; return;; /* otherwise*
<look:> if n k ≥∃j > 1 |
   numnodes(odesc(top,j)) lt numnodes(odesc(top,j-1)) or
   seqno(odesc(top,j)) lt seqno(odesc(top,j-1))
   then j = 1;;
<adv:> advance(odesc(top,j),more);
   set = [+: j<k≤#odesc{top}] nodes(odesc(top,k)) orm    nℓ;
   if n more then go to movenode;;
   /* else */ hang(top,j+1,set); return;
<movenode:> if set eq nℓ then go to backup;;
   setl=nodes(odesc(top,j));
   newtop = ∋set;
   <numnodes(newtop),seqno(newtop)> = <#setl+1,1>;
   odesc(top,j) = newtop; hang(newtop,1,setl);
   hang(top,j+1,set less newtop);
   return;
<backup:> if j eq 1 then more = f; return;;
   /* else */ k = j-1; go to look;
end advance;

define  hang(top,j,set);
k = j;
(while odesc(top,k) ne Ω doing k = k+1;)
   odesc(top,k) = Ω;;
k = j;
(∀nod ∈ set)
   odesc(top,k) = nod;  k = k+1;
   <numnodes(nod),seqno(nod)> = <1,1>;
   odesc(nod) = Ω;
end ∀nod;
return;
end hang;

definef nodes(top);
if numnodes(top) eq 0 then return {top};;
return [+: 1 ≤ j ≤ #odesc{top}] nodes(odesc(top,j)) with top;
end nodes;
```

4. Algorithms for Permutations.

A *permutation* of a set s of n objects is a one-to-one mapping of s into itself. The set of permutations of s forms a group, indeed, the prototype of all groups; the combination and inverse being given as follows:

```
definef f c g; return{<x, g(f(x))>, x ∈ hd[f]}; end c;
definef inv f; return{<x(2), x(1)>  x ∈ f};     end inv;
```

Permutations can conveniently be represented in the so-called *cycle form*. The cycle form of a permutation f of s is a family of disjoint sequences, collectively covering s, such that f(x) is always the next element in sequence after x, unless x is the last element of a sequence, in which case f(x) is the first element of the same sequence. A SETL algorithm for putting a permutation into cycle form may then be written as follows:

```
definef cycform(f);
s = hd[f]; cycs = nl;
(while s ne nl) elt from s; cyc = <elt>;
   (while f(elt) is e ∈ s doing elt = e;)
        cyc = cyc + <e>;
        s = s less e;
   end while f;
   cyc in cycs;
end while s;
return cycs;
end cycform;
```

Since the cycle form of a permutation represents the permutation in a condensed and structurally revealing manner, it is useful to be able to perform the basic operations of combination and inversion directly on the cycle form of a permutation. Inversion is easy: we merely reverse every cycle. In SETL this is:

```
definef invc cycs;
return {[+: #c>n>1]<c(n)>, c ∈ cycs};
   invc;
```

The existence of the cycle form of a permutation shows that ar-- permutation may be written as a product of 'cyclic' permutations i.e., permutations of the special form a → b, b → c, ..., d → e, e→a. (The notation (abc...de) is often used for a permutation of this special sort.) A number of algorithms for the inversion of permutations 'in place', (i.e., without the use of extra storage space; assuming that the permutation is represented by a permuted array of values) implicitly make use of the cyclic decomposition of a permutation. The simplest of these inversion algorithms rather resembles the algorithm for putting a permutation into cycle form.

```
define  cyclinv(f); s = hd[f];
(while s ne nl) elt from s; next = f(elt);
   (while next ∈ s) s = s less next;
        <f(next),elt,next> = <elt,next,f(next)>;;
   f(next) = elt; /* closing the loop */
end while s;
return;
end cycinv;
```

A very short but quite enigmatic algorithm serving the same purpose, due to Boothroyd, may be written as follows.

```
define boothinv(f);
s = hd[f];  heads = s;
(∀p ∈ s) q = p;
   (while n q ∈ heads) q = f(q);;
   r = f(q); f(q) = f(r); f(r) = p;  heads = heads less r;
end ∀p;
return;
end boothinv;
```

The analysis of this very short algorithm is surprisingly complex, and may be given as follows. Let f be a cyclic permutation, which we may think of as shifting a group of elements arranged in a circle, each being shifted to the next position around the circle. Flag h element of s as a "head". Then, *process* the elements of s, in any

random order, as follows. Take any as yet unprocessed element p, find the first element q in the sequence p, f(p), ..., $f^k(p)$ which is still flagged as a head; drop the flag of r = f(q), and put f(q) = f(r), f(r) = p.

To follow the action of the algorithm, we take the set s to be divided into a set of *runs*, where each run consists of a sequence of elements (in their original circular order) beginning with an element flagged as a head, up to but not including the next element flagged as a head. Then note that by induction we have the following:

i. By definition, the first element of a run is flagged, all others are unflagged.

ii. The last element of a run is unprocessed; the others have all been processed.

iii. For all p of a run but its first element, f(p) is the previous element in circular order; if p is the first element of a run, f(p) is the first element of the next succeeding run (in circular order).

All these remarks hold initially, all runs initially being of unit length. Since by (ii) every unprocessed element p is the last element of a run, our procedure will always find the head q of the same run, and by (iii) the head r = f(q) of the next run in circular order. By putting f(q) = f(r), f(r) = p , and dropping the flag of r we join two runs into a single run, preserving the properties i , ii and iii.

This proves that Boothroyd's process works for every cyclic permutation; since every permutation can be decomposed into cyclic permutations, it must work for every permutation.

Next we consider algorithms for the multiplication of permutations given in cyclic form. Suppose that we have a sequence of cyclic permutations

(1) (...abcd...) $\underline{c}$ (...efgh...) $\underline{c}$ ... $\underline{c}$ (...ijkℓ...)

to be multiplied together. (We continue to assume a 'left-to-right' convention; that is, the first of these maps to be applied to a set is the *leftmost*.) Then the image of an element a under the product map is obtained by finding the leftmost occurrence of a,

and the element b which follows it; then the next occurrence of k further to the right, and the element c which follows it, etc., until the right-hand end of the sequence (1) of cycles is reached. Each occurrence of a symbol x in the product (1) will be used in such an image-finding subprocess only once; thus if we mark the occurrences that have been used, and continue the image-finding subprocess as long as any positions in the sequence 1 are still unmarked, we can multiply permutations in cycle form without having to know the set of elements on which these permutations act. The algorithm is most conveniently expressed if, at the end of each cycle, we repeat the first element of the cycle, and mark the corresponding position as used.

The following small example will clarify the procedure just outlined. Suppose that the permutations (1 2 3)(4 5) and (3 4 2) are to be multiplied. We prepare for multiplication by establishing the sequence

$$1\ 2\ 3\ \underline{1}\ 4\ 5\ \underline{4}\ 3\ 4\ 2\ \underline{3}$$

('marked' elements are indicated by underbars). In a first left-to-right pass over the sequence, we determine that $1 \to 2 \to 3$; thus we obtain the result-fragment

$$(1\ 3$$

and the additional markings

$$\underline{1}\ 2\ 3\ \underline{1}\ 4\ 5\ \underline{4}\ 3\ 4\ \underline{2}\ \underline{3}\ .$$

Next we find that $3 \to 1$; completing our first cycle

$$(1\ 3)$$

and giving the markings

$$\underline{1}\ 2\ \underline{3}\ \underline{1}\ 4\ 5\ \underline{4}\ 3\ 4\ \underline{2}\ \underline{3}\ .$$

An additional left-to-right pass shows that $2 \to 3 \to 4$, giving the output fragment

$$(2\ 4$$

and the marking

$$\underline{1}\ \underline{2}\ \underline{3}\ \underline{1}\ 4\ 5\ \underline{4}\ \underline{3}\ 4\ \underline{2}\ \underline{3}\ .$$

Next we find that 4 → 5, bringing us to the output fragment

(2 4 5

and the marking

1 2 3 1 4 5 4 3 4 2 3 .

Finally we see that 5 → 4 → 2 , at which point all components of our sequence are marked, and we have the result permutation (1 3)(2 4 5).

The following SETL algorithm, which assumes that an ordered tuple *seqperms* of permutations in cycle form is given, uses the procedure just explained.

```
definef multall(seqperms);
/* first make single sequence with repetitions */
seq = nult;  marked = nl;
(∀perm(m) ∈ seqperms, cyc ∈ perm)
   (∀elt(m) ∈ cyc) seq(#seq+1) = elt;;
   seq(#seq+1) = cyc(1); /* repeating the first element */
   (#seq) in marked /* marking the repeated elements */
end ∀perm;
result = nl;
(while ∃e(m) ∈ seq | n m ∈ marked) /* start new output cycle */
   cyc =  <seq(m)>;   elt=seq(m+1);
   loc = m+1;  m in marked;
   <loop:>
   (while loc < ∃n < #seq | seq(n) eq elt and n  n ∈ marked)
        n in marked;
        elt = seq(n+1); loc = n+1;
   end while loc;
   if elt ne cyc(1) then
        cyc(#cyc+1) = elt; loc = 1; go to loop;
   else cyc in result;
   end if;
end while ∃e(m);
return result;
end multall;
```

We now give another algorithm, which 'in one pass' multiplies permutations given in cycle form. The algorithm exploits the following facts:

i. If * is an object not appearing in a cycle, then the cyclic permutation (ab...cdef) is the following product of maps

$$(f \to *) \ \underline{c} \ (e \to f) \ \underline{c} \ (d \to e) \ \ldots \ \underline{c} \ (a \to b) \ \underline{c} \ (* \to a) \ .$$

ii. If *map* is any mapping, then the composition

$$map' = (x \to y) \ \underline{c} \ map$$

is defined by map'(x) = map(y); map'(z) = map(z) if z is different from x.

The SETL code for this algorithm, which as might be imagined works from right to left, is as follows:

```
definef multall2(seqperms);
map = nl;
(#seqperms ≥ ∀n ≥ 1, cyc ∈ seqperms(n) | #cyc gt 1)
   mapstar = if map(cyc(1)) is x ne Ω then x else cyc(1);
   (1 ≤ ∀i < #cyc)
        map(cyc(i)) = if map(cyc(i+1)) is x ne Ω then x elsecyc(i+1);
   end ∀i;
   map(cyc(#cyc)) = mapstar;
end ∀n;
return cycform(map);
end multall2;
```

5. A Data Compaction Algorithm.

In any computer system, storage space, and especially high-quality storage, is available only in limited amounts. For this reason the problem of table anddata compaction, i.e., the problem of encoding a given mass of data into the most compact form possible, is an important one. The same problem arises also in connection with the transmission of data; in order to maximize the rate at which useful information is transmitted, we wish to make use of an encoding which reduces to a minimum the number of bits which must be transmitted. This compression problem has been much studied, and has generated a vast literature; we shall in the present short section touch on only one of the interesting ideas which have been developed. Generally speaking, data is compressed by discovering and exploiting regularities in it; the most drastic compressions will come when a large table of data can be replaced by a short routine capable of calculating all its entries. Even where this is not possible, any statistical regularity which the data exhibits may be used to secure compression. The algorithms we shall now exhibit, those for *Huffman coding*, make use of this idea.

The setting assumed is as follows. Text, consisting of a stream of characters, is to be encoded. The various possible characters occur with differing relative frequencies, the expected frequency of character c being given by *freq(c)*.

It is then advantageous to assign a binary code to each character in such a way that the most probable characters receive short codes, while the least probable characters receive long codes. In this way we may on the average expect a text to be represented compactly. Huffman's specific technique is as follows: take the two characters c,d of smallest frequency, and hang them as left and right branches from a newly created node n, whose heuristic meaning is 'either c or d'. Remove c and d from the set of characters and insert n, taking its frequency to be the sum of that of c and d. Repeat this operation until only a single character remains, in the process growing a tree, the so-called *Huffman tree* of the set of characters. The code for a character is then its address in this tree, where 'go down to the left' is represented by a binary 0, and 'go down to the right' is represented by a binary 1. The decoding process

to be applied to a stream of characters represented in Huffman coded form by a binary sequence is then clear: Start at the top of the tree, and proceed downward to a twig, in the manner directed by the binary sequence. When a twig is reached, take the character found there as the next symbol of a decoded sequence, jump back to the top of the tree, and repeat.

In SETL, the Huffman-tree and encoding-table build routine appears as follows:

```
definef huftables(chars,freq);
work = chars; wfreq=freq; ℓ = nℓ; r = nℓ;
(while #work gt 1)
   c1 = getmin work; c2 = getmin work; n =newat;
   ℓ(n) = c1;  r(n) = c2;
   wfreq(n) = wfreq(c1) + wfreq(c2); n in work;
end while;
code = nℓ; seq = nulb; walk(∃work is top);
return <code,ℓ,r,top>;
end huftables;
definef getmin set; /* freq is global */
<keep,least> = <∃set is x, freq(x)>;
(∀x ∈ set)
   if freq(x) lt least then <keep,least> = <x,freq(x)>;;
end ∀x;
keep out set;  return keep;
end getmin;
define  walk(top); /* recursive tree-walker which builds up
        address of each twig */
/* code,  seq are global */
if ℓ(top) ne Ω then
   seq = seq + f;  walk(ℓ(top));
   seq = seq + t;  walk(r(top));
else /* at twig */ code(top) = seq;
end if;
seq = seq(1: #seq-1);
return;
end walk;
```

Next, the decoding process. Given a sequence *bitseq* of bits, produce a sequence *cseq* of characters by the following decoding algorithm.

```
definef cseq(huftables,bitseq);
<-,ℓ,r,top> = huftables;
output=nulc; node = top;
(∀b(n) ∈ bitseq)
   if ℓ(node) eq Ω /* so that we are at twig */
        then output(#output+1) = node; node = top;
   else node = if n b then ℓ(node) else r(node);
   end if;
end ∀n;
return output;
end cseq;
```

The basic Huffman-coding technique that we have outlined can be refined in a number of ways. Instead of encoding single characters, we can encode pairs or triples of characters using the Huffman technique. Instead of using a fixed set of frequencies, we can collect separate tables showing the relative frequency which each character will have in known preceding contexts, e.g. the relative frequency which c will havein those cases when it follows immediately after a given character d. This information can then be used to build up a set of Huffman tables, one for each context that we wish to distinguish. During the decoding process, we will of course know the preceding context of each symbol s we are attempting to decode, having decoded the symbols determining this context before we begin trying to decode s. We leave it to the reader to work up generalized Huffman coding and decoding algorithms incorporating these more sophisticated possibilities.

6. An algorithm for the SETL input-read process.

In the present section, we shall use SETL to describe a central part of the SETL input routine. The algorithm to be given furnishes an interesting example of use and advantages of the flow statement. It will also provide us with good ground on which to compare other proposed programming styles (including a type of scrupulously goto-less style) with the more familiar pragmatic style which describes control flow by using a judicious mixture of if-then and go-to statements.

The algorithm to be presented corresponds to the single 'parsing' operation performed during input. It calls a lower level routine, *tokread*, to get successive tokens from an input file, and uses a table *spekind* from which a numerical attribute *tokind* is obtained for each token. Tokens are classified into three categories: atoms (for which tokind = Ω), composite object delimiters (opening and closing symbols for sets and tuples) and separators: blank, comma, E-O-R and E-O-F.

Two stacks, *partstack* and *termstack*, are used. The first, *partstack* stores partly-built composite objects; *termstack* stores opening delimiters. Whenever a new opener is found, a corresponding null object (nult or nl) is added to the top of *partstack*. When a closing delimiter is encountered, a matching opener has to be found on the top of *termstack*; at this point the composite item being scanned (the current top of *partstack*) is complete.

This completed item is either returned from the *read* routine (if it is the only item present in *partstack*) or it is added to the next lower element on *partstack* which becomes the current composite object being built.

Further details of the algorithm can easily be gleaned from the SETL test that follows. For purposes of comparison, we give the algorithm in three SETL versions. In each version (flow-statement style, style using nested ifs without go-to statements, and style using nested ifs plus go-to's) the following macros are used:

```
+* top(x) = x(#x) **
+* newtop(x) = x(#x+1) **
+* set = 1 **  +* opentuple=2 **  +* closeset = 3 **
+* closetuple = 4 **  +* slash=5 **  +* comma=6**  +* ef = 7**
```

The table *spekind* is then given by the following mapping:

```
<<'{',set>,  <'<', opentup>, <'}',closeset>, <'>', closetuple>,
<'/', slash>, <',', comma>,< '$', ef>>
```

Our first version of the input algorithm, using the flow statement, is then as follows:

```
definef read(file);
/* SETL read routine */
partstack = nult;  termstack = nult;
start:  tok = tokread(file);  tokind=spekind(tok);
        flow                              special?
             startnew?                                  stackempty?
    startit+                      termin?          returnit,   enterit+
    start,       matches?                  comma?              start,
              first?  error,   stackempty?          slash?
         retfirst,  endit+     error,   start,   stackempty? crashterm,
                    start,                  start,  errprint;


special:=     tokind ne Ω;
stackempty:=  termstack eq nult;
returnit:     return tok;
enterit:      if top(termstack) eq set then top(partstack)=top(partstack)
                        with tok;
                        else top(partstack) = top(partstack)+<tok>;;
startnew:=    tokind le opentuple;
termin:=      tokind le closetuple;
matches:=     top(termstack) eq (tokind - opentuple);
startit:      newtop(partstack) = if tokind eq set then nl else nult;
              newtop(termstak)  = tokind;
```

```
first:  = (#termstack) eq 1;
retfirst: return partstack(1);
endit:  composite = top(partstack); top(partstack)=Ω;
        if top(termstack) eq set then top(partstack)=top(partstack)
                with composite;
            else top(partstack) = top(partstack) + <composite>;
            top(termstack) = Ω;
comma:    = tokind eq comma;
slash:    = tokind eq slash;
crashterm:  print 'end of file on attempt to read'; crash;
        end flow;
        go to start;
error:  skipover(file);  /* reads to next slash mark */
errprint: print   'illegal configuration detected on attempted
        file read.  skipping past next slash mark';
        return Ω;
        end read;
```

Next we give a goto-free version of the same algorithm. Notice that the outer loop (corresponding to the statement: go to start; in the preceding version) is here coded using a while statement with a dummy variable.

```
definef read(file);
errmsge = ' illegal configuration detected on attempted file read.
          skipping past next slash mark.';
partstack = nult;  termstack = nult;
(while t)
   tok = tokread(file);
   tokind = spekind(tok);
   if tokind eq Ω then
      if termstack eq nult then return tok;;
      /* else */ topp = top(partstack);
      if top(termstack) eq set then
         tok in topp;
      else
```

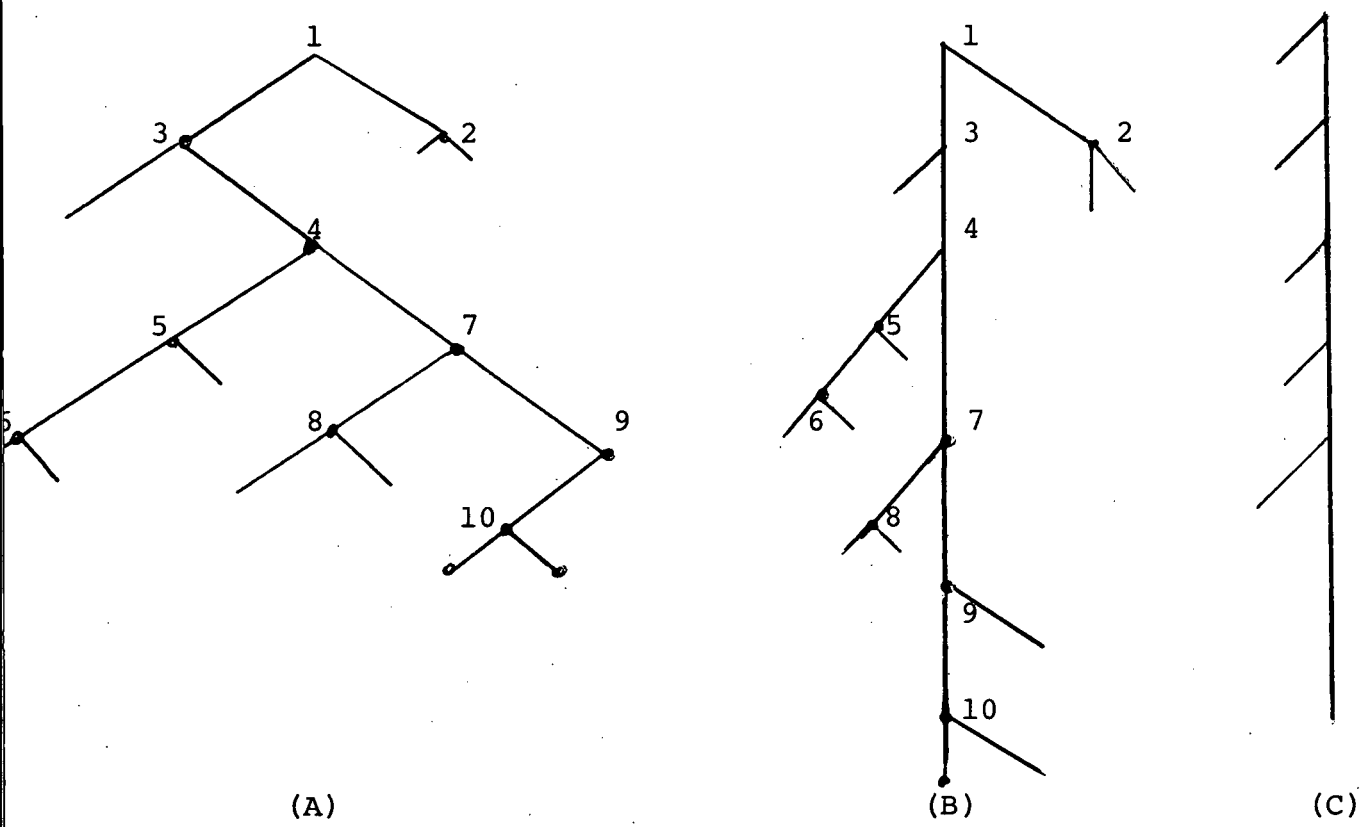
```
        newtop(topp) = tok;
      end if top;
      top(partstack) = topp;
   else if tokind le opentuple then
        newtop(partstack)=if tokind eq set then nl else nult;
        newtop(termstack) = tokind;
   else if tokind le closetuple then
        if top(termstack) ne(tokind - opentuple) then
           skipover(file);  print errmsge;  return Ω;
        else if(#termstack) eq 1 then return partstack(1);
        else composite = top(partstack);
             top(partstack) = Ω;
             topp = top(partstack);
             if top(termstack) eq set then
                composite in topp;
             else
                newtop(topp) = composite;
             end if top;
             top(partstack) = topp;
             top(termstack) = Ω;
        end if top(termstack);
   else if tokind eq comma then
          if termstack eq nult then
            skipover(file);  print errmsge;  return Ω;;
   else if tokind eq slash then
          if termstack ne nult then print errmsge;  return Ω;;
   else print  'end of file on attempt to read';  crash;
   end if tokind;
end while;
end read;
```

Our third version of the same algorithm, which may be called the "pragmatic" version, can be coded as follows:

```
     definef read(file);
     /* SETL read routine, transcribed to avoid flow statement
        by use of go-to statement */
     partstack = nult;  termstack = nult;
start:  tok = tokread(file);  tokind = spekind(tok);
        if tokind ne Ω then go to startnew;;
        if termstack eq nult then return tok;;
        topp = top(partstack);
        if top(termstack) eq set then tok in topp;
             else newtop(topp) = tok;
        end if top;
        top(partstack) = topp;
        go to start;
startnew: if tokind gt opentuple then go to termin;;
        newtop(partstack) = if tokind eq set then nl else nult;
        newtop(termstack) = tokind;
        go to start;
termin: if tokind le closetuple then go to matches;;
        if tokind eq comma then go to stackempty;;
        if tokind eq slash then go to stackempty2;;
        print 'end of file on attempt to read';  crash;
stackempty: if termstack ne nult then go to start;;
error:  skipover(file);  /* reads to next slash mark */
errprint: print 'illegal configuration detected on attempted file read
             skipping past next slash mark';
        return Ω;
matches: if top(termstack) ne (tokind-opentuple) then go to error;;
        if (#termstack) eq 1 then return partstack(1);;
        composite = top(partstack); top(partstack) = Ω;
        topp = top(partstack);
        if top(termstack) eq set then composite in topp;
             else newtop(topp) = composite;
        end if top;
        top(termstack) = Ω;
        top(partstack) = topp;
        go to start;
stackempty2: if termstack eq nult then go to start;;
        go to errprint;
        end read;
```

The following comparative remarks concerning these three versions of one and the same algorithm are worth making:

a) The flow version has the layout diagrammed by (A) below. It is by far the eastest version to read. It is also the easiest to code, and the flow tree (whose skeleton can be represented as (A)) is an extremely useful starting point for transcribing the algorithm into one of the other versions, or into a lower level language.



(A) (B) (C)

b) The goto-less version is obscure, and borders on the undecipherable. It is very hard to code. The flow-tree can be of help in producing the goto-less version, especially if "stretched" along the longest decision path, so that the nodes of the main if-then-else statement become apparent as in (B). In any case, it is clear that the if-then-else statement is mainly suited for trees with little or no lateral growth (ZPG trees?) as in (C), and not for 'fertile' trees like (A).

c) The third pragmatic version of our algorithm is somewhat harder to read than the flow version, but does not involve substantially

more non-local eye motion (especially in locating actions to be performed) although the physical proximity of logically related decisions has been lost. It is however much easier to read than the structured version. The flow-tree translates easily into the appropriate go-to statements.

An attractive mixture of go-to and if statements might involve the use of if-then-else statements on ZPG branches of the main tree (as in the block following the label *termin* in the third program given above, corresponding to node no. 4 in (A)), and the use of go-to statements to enter each such subtree.

7. Parsing and Other Miscellaneous Compiler-Related Algorithms.

In the present section we will describe a set of general algorithms which serve to define efficient table-driven parsers for a wide class of programming languages. Generally speaking, parsing constitutes (after a lexical scan) the first stage of a total compilation process. A lexical scanner breaks an input stream of characters into a string of *tokens*. The parser then associates a tree, called the *parse tree*, with this string of tokens. This tree is valuable for the following reason: it establishes a fixed system of *tree addresses* for the tokens of the sentence parsed, and does this in such a way that *tokens playing given roles within a sentence will always be found at fixed corresponding tree addresses*. For example, in statements, statement-type determining keywords will always be found at certain particular tree nodes; in expressions, the *central operator sign*, i.e., that sign denoting the operation to be performed last, will always be found at certain particular tree nodes, etc. This is of course not the case for the string of tokens originally representing a sentence.

The first parsing technique we shall describe is the 'nodal span' method of Cocke, Younger, and Earley; a detailed discussion of this algorithm will be found in Cocke and Schwartz, *Programming Languages and Their Compilers*, second revised version, New York University, April 1970, section 4.6. In contrast to most other parsing procedures, this method is highly stable in the presence of ambiguity; in particular, it can be applied to any context-free grammar whatsoever; moreover, the method can readily be generalized to apply to grammars more flexible than simple Backus grammars. It can also be adapted to grammars in which a central group of exactly correct rules is surrounded by a penumbra of partly incorrect rules, and in this adaptation can be used to assign a quantitative 'degree of grammaticalness' to sentences which are not quite grammatical. It also suggests interesting methods for the parsing of structures, such as plane figures, which are more general than linear strings.

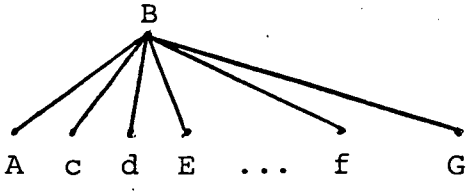
In describing this algorithm, the notion of a *context free* or *Backus* grammar is convenient. Such a grammar is defined by givin

i. An alphabet of tokens, called the *alphabet* of the grammar; the symbols in this alphabet are distinguished into two classes, *intermediate* symbols and *terminal* symbols. We shall often indicate intermediate sybmols by capital letters, terminal symbols by small.

ii. For each intermediate symbol B, a finite set of *productions* of the form

(1) $$B \rightarrow AcdE...fG \ ,$$

where the right-hand side of a production may be any finite sequence of intermediate and terminal symbols. Heuristically, the inclusion of the production (1) in a grammar means that the symbol B, whenever it appears in a sentence of the language which the grammar describes, can be replaced by a string AcdE...fG . Conversely, this means that it is possible to cover the string AcdE...fG by a treelet



when sentences are being supplied with parse trees.

iii. One intermediate symbol of the grammar must be designated as its *root* or *sentence* type. Valid sentence-forms are then all those strings of symbols which can be produced from the root symbol by the successive application of productions of the grammar; valid sentences are all those valid sentence-forms containing terminal symbols only.

Having said this, we prepare to present a SETL program for the nodal span parsing algorithm by giving a brief review of this algorithm's structure. The algorithm accepts an input sequence of tokens and a context-free grammar, and parses the tokens according to the grammar. The grammar may be ambiguous or unambiguous, but, in order to simplify our exposition, we take it to be given in a standardized "reduced" form in which each production of the grammar either has the form A → a or the form A → BC, A, B, and C deno intermediate symbols of the grammar, and a denoting a terminal symbol of the grammar. In this situation we call a triple (pAq)

consisting of two integers p, q and an intermediate grammatical mbol A a *span*. We say that this span (pAq) is *present* in the input if the string of tokens extending from the pth through the q-1-st position in the input string can be generated from the symbol A using products of the grammar. For q = p+1 this will clearly be the case if and only if some production A → a generates the p-th character of the input string; for q > p+1 this will equally clearly be the case if and only if there exists some r, p < r < q, and two intermediate symbols B, C of the grammar such that both spans (pBr) and rCq) are present in the input, and such that A → BC is a valid production of the grammar. In this case we say that the triple (BrC) is a valid *division* of the span (pAq); the set of all divisions of a span is called its *division list*. If A_0 is the *root* symbol of the grammar, an input string I of length n is grammatical if and only if the *root span* ($1A_0(n+1)$) is present in it. In this case, producing the collection of all division lists for all spans present in I may be regarded as equivalent to parsing I; if these lists are available the ordinary "parse tree" of I can be obtained in a perfectly direct way by dividing the basic span into parts in all the ways indicated by its division list, dividing all these parts into subparts using their division lits, etc. until spans of unit length are reached.

It is also clear that not all spans present in the input, but only those spans which are obtained from the basic span by this process of division, are *relevant* to the final analysis of the input. Other spans present in the input are in effect merely false starts never resulting in a complete parse. The set of division lists belonging to the narrower collection of spans relevant in this sense is called the *cleaned* division list for the parse of the input string.

The above remarks should make clear the structure of our parsing algorithm. We proceed from left to right, accumulating spans present in the input string. On encountering the n-th symbol a of the input string, we add a span (nC(n+1)) to our collection whenever the existence of a production C → a justifies this.

ly added spans (rCq) are then "processed" by locating all spans (pBr) for which the existence of a production A → BC in the grammar

will yield a new span (pAq). Each time two spans are combined in this way to give a span (pAq) we make an appropriate entry on the division list of the span (pAq). When as many spans as possible have been produced by this combination process, we go on to consider the next input token in turn, and so forth to the very end of the string of input tokens. When the end of the string of input tokens has been reached we check to see whether or not the root span ($1A_0(n+1)$) is present; if this basic span is present, we prepare a cleaned set of division lists by the division process described above.

The SETL program given below carries out the steps just outlined, determining at the same time whether the given input has an ambiguous or unambiguous parse; the criterion for ambiguity is simply that at least one of the division lists in the cleaned set should contain more than one element. A few additional preliminary remarks will cast light on the details of the program which follows. Spans are represented as ordered triples <q,A,p> composed of two integers p, q and an intermediate grammatical symbol A. For technical reasons the larger integer q is placed first and the smaller second. The family of division lists belonging to such spans is maintained as a collection of quadruples <<q,A,p>, r,B,C> associating with each span all its divisions (BrC). The context-free grammar according to which the input is to be parsed is taken in the program which follows to be the collection of all triples <B,C,A> corresponding to productions A → BC of the grammar. A function *syntypes*, which maps each character a of the input string onto the set of all intermediate symbols A for which there exists a production A → a, is also assumed in the following program.

Here then is a SETL version of the basic algorithm for parsing by the method of nodal spans.

```
define nodparse(input,gram,root,syntypes,spans,divlis,amb);
todo = nl; divlis = nl; spans = {<2,s,1>, s ∈ syntypes{input(1)}};
(1 < ∀n ≤ #input)
   todo = {<n+1,s,n>, s ∈ syntypes{input(n)}};
   spans = spans + todo;
```

```
    (while todo ne nl)
    next from todo;
    <end,typ2,mid> = next;
         (∀spend ∈ spans{mid}, type∈gram{hd spend is typ1,typ2})
              newsp = <end,type,spend(2)>;
              <newsp,mid,typ1,typ2> in divlis;
              if n newsp ∈ spans then
                   newsp in spans;  newsp in todo;
              end if;
         end ∀spend;
    end while;
end ∀n;
/* check on grammaticalness */
if n (<#input+1,root,1> is topspan) ∈ spans then
    <spans,divlis,amb> = <nl,nl,f>; return;
end if;
/*  else clean up set of spans and determine ambiguity */
spans = nl;  amb = f;  getdescs(topspan);
/* clean division list */
divlis = {d ∈ divlis | hd d ∈ spans};
return;
end nodparse;
define getdescs(top); /* auxiliary tree-walker */
/* divlis, spans, amb are global */
 if top∈spans then return;; /*since descendants have already
                                  been added */
top in spans;
if (#divlis{top}) gt 1 then amb = t;;
<end,-,start> = top;
(∀x ∈ divlis{top})
    <mid,typ1,typ2> = x;
    getdescs(<end,typ2,mid>);
    getdescs(<mid,typ1,start>);
end ∀x;
return;
end getdescs;
```

In unfavorable cases, the nodal span parsing method may generate reasonably large numbers of spans not relevant to the final analysis of an input string. J. Earley has introduced an interesting modification of nodal span parsing which improves its performance in this regard. In Earley's method, one is always sure, as one scans from left to right over an input string $S = t_1\ t_2\ \ldots\ t_m$ of tokens, that a span pAq+1 will never be formed unless there is some continuation $t_1 \ldots t_q t'_{q+1}\ t'_{q+2}\ \ldots$ of the inital portion $t_1\ t_2\ \ldots\ t_q$ of S to whose analysis pAq+1 is relevant. Thus one never forms spans which are not 'relevant to as much of the input as has been scanned'.

The method is this. Immediately before scanning the n-th token of S, one calculates a set, called *startat(n)* in the algorithm which follows, which consists of all those intermediate symbols of the grammar Γ being considered which can validly follow the part of S already scanned. The rule for forming this set is simple and iterative: If there exists a span pBn, and intermediate symbols A and C, such that A→BC is a production of Γ and such that A ∈ *startat(p)*, then any sumbol D for which there exists a string DEF... derivable from C by productions of Γ belongs in *startat(n)*. Having constructed these sets, we use them in the remainder of the algorithm to control the formation of additional spans, always applying the following test: no span pAq is to be formed for which A ∈ *startat(p)* is false.

In other respects, Earley's procedure, for which we now give a SETL algorithm, is very close to the nodal span parsing algorithm set forth above.

```
/* auxiliary macro block used below */
macro makenewtodofrom(next);
<end,typ2,mid> = next;
(∀spend ∈ spans{mid}, type∈gram{hd spend is typ1,typ2}
                    | type ∈ startat(spend(2)))
    newsp = <end, type, spend(2)>;
    <newsp,mid,typ1,typ2> in divlis;
    if n newsp ∈ spans then
        newsp in spans; newsp in todo;
    end if;
```

```
end ∀spend;
  dm;
   note use below of the transitive closure routines given
   on pp. 119 ff */
define earleyparse(input,gram,root,syntypes,spans,divlis,amb);
/* first calculate the fixed relationships needed */
begins = {<x(3),x(1)>,x ∈ gram};
descends = begins + {<x(3), x(2)>, x ∈ gram};
symbs = close (descends,{root}); ultbegin = closef(begins,symbs);
/* the above need not be repeated unless the grammar is changed */
todo = nl; divlis = nl; startat =  <ultbegin{root}>;
    spans = {<2,s,1>, s∈syntypes(input(1))|s∈startat(1)};
(1 < ∀n ≤ #input)
    /* calculate startat(n) */ pops = nl;
    (∀spanend  ∈ spans{n})
        <typl,beg> = spanend;
        pops=pops+{hd x, x∈gram{typl}|x(2)∈startat(beg)};
    end ∀spanend;
    startat(n) = ultbegin[pops];
    /* now go on to form spans */
    todo ={<n+1,s,n>, s∈syntypes(input(n))|s∈startat(n)};
    spans = spans + todo;
    (while todo ne nl)
        next from todo; makenewtodofrom(next);
    end while;
end ∀n;
    /* check on grammaticalness */
if n (<#input+1,root,1> is topspan ε spans then
    <spans,divlis,amb> = <nl, nl, f>;  return;
end if;
/* else clean up set of spans and determine ambiguity */
spans = nl;  amb = f;  getdescs(topspan);
/* clean division list */
divlis = {d ε divlis |(hd d) ε spans};
return;
end earleyparse;
   a copy of the subroutine 'getdescs' occurring on page 162 is
    also required here */
```

The next parsing method which we shall describe, the so-called *generalized precedence method* of McKeeman and others, represents a very considerable extension of various simpler and more familiar precedence parsers.

The method used is interesting, and can yield rather fast parsers. It has however a number of disadvantages. In particular, a carefully thought out extension of the precedence parse technique is needed if high-quality diagnostics are to be produced.

The key idea of generalized precedence parsing is as follows. Parsing ascribes a tree to a string of tokens; more precisely, it attaches the tokens of an input string to the twigs of a parse tree. For example, we may represent the parse of an expression ...+a*b-c*d-... as follows:

If a token (such as a or c in our example) is attached to a left-hand twig of the parse tree, we say that the parse *leans right* from the token; if a token (such as b or d in our example) is attached to a right-hand twig, we say that the parse *leans left* from the token. It is intuitively clear that if we can always tell which way the parse leans at each token, we can reconstruct the entire parse tree, using the following method: Scan the input string, and find the first token t at which the parse leans left. Then find the last token t' preceding t at which the parse leans right. The string of tokens between t' and t must all be descendants of some single node in the parse tree. Building a treelet to which this string of tokens is attached, and replacing, or so to speak *condensing* this string of tokens into a single token representing the root of the treelet, we obtain a shorter string, to which the same process can be applied again. Repeated condensations will eventually reduce the whole of a token string to a single symbol; in the process, its parse tree will have been built.

Next observe that we can generally expect to be able to duce the direction in which the parse tree of a string leans from a given token t by examining a very few tokens in the immediate neighborhood of t. This is the basis for the simple precedence parse; a language can be parsed by the simple precedence method if its parse trees always lean in the direction of whatever nearby operator sign has highest precedence. Even for languages too complex for this ultra-simple rule to work infallibly, a slightly more sophisticated variant of the same method may be feasible; we call languages for which this is the case *extended precedence languages*.

In the logical setting established by a context-free grammar Γ we may give formal definitions capturing the ideas concerning extended precedence which are set forth above. These definitions are as follows. Let

$$\alpha = a_1 \ldots a_n, \quad \beta = b_1, \ldots, b_m, \tag{2}$$

be two strings of symbols, terminal or non-terminal, of the alphabet of Γ. We define three relationships, $\alpha \cdot\!> \beta$, $\alpha \doteq \beta$, $\alpha <\!\cdot \beta$ between such strings. The definitions are as follows:

i. $\alpha \cdot\!> \beta$ (heuristically: the parse should lean toward α from the last symbol of α) if $\ldots a_1 \ldots a_n\ b_1 \ldots b_m \ldots$ can occur within a sentence σ, in whose parse tree a_n is attached to the rightmost twig of some subtree.

ii. $\alpha <\!\cdot \beta$ (heuristically: the parse should lean toward β from the first symbol of β) if $\ldots a_1 \ldots a_n\ b_1 \ldots b_m \ldots$ can occur within a sentence σ, in whose parse tree b_1 is attached to the leftmost twig of some subtree, while a_n is not attached to the rightmost twig of a subtree.

iii. $\alpha \doteq \beta$ (heuristically: the parse is balanced at the last symbol of α) if $\ldots a_1 \ldots a_n\ b_1 \ldots b_m$ can occur within a sentence σ, in whose parse tree a_n and b_1 are both attached to twigs with a common parent node.

If at most one of these three relationships holds between α and β, we call the pair α,β an *unambiguous context*. (If none holds, we call α,β an *impossible context*; it is easy to see that in this case $\ldots a_1 \ldots a_n b_1 \ldots b_m \ldots$ can never occur within a grammatical sentence.) If the language defined by Γ is such that whenever α is a sequence of symbols of length n and β is a sequence of symbols of length m it follows that α,β is an unambiguous context, then we say that is an *n,m-generalized precedence grammar*. In this case, the generalized precedence parsing method outlined above can be used to reconstruct the parse tree of any sentence valid according to Γ.

Parsing by this scheme will plainly involve the use of tables which state the relationships which hold between sequences α and β of symbols. We shall now present a SETL algorithm which, given a grammar (and given limits n,m for the lengths of left- and right-hand contexts to be considered) will produce such tables. The basic idea of the algorithm is quite simple. We start with a pair of single symbols, and determine whether or not these 1-token strings constitute an unambiguous context. If they do not, we add one symbol of additional context, first on the left, then on the right, looking always for unambiguous contexts. This will generate a sequence of tables which collectively contain the required precedence information. If,when we have reached the maximum length-limit for contexts to be considered,ambiguities still remain, then it follows that the grammar being considered is not an n,m-generalized precedence grammar.

The algorithm which follows immediately below is a quite straightforward elaboration of this idea. It works with left- and right-hand contexts of the lengths (1,1), (2,1), (1,2), (2,2), (3,2), (2,3), (3,3) in sequence, up to a stated maximum length. A data-structure *table(seqa,seqb)*, which is progressively built up, records values 1, 2, and 3, depending on the relation

of precedence that holds between the sequences seqa and seqb characters; the value 0 is used if this relationship is ambiguous. Amgibuous pairs are held temporarily in a set *ambig*.

An output flag *definite* is set to t if the situation analyzed has the *lmax, rmax* precedence property; to f otherwise. The definitions of the basic relationships *seqa* •> *seqb* , etc., given above convert very directly into algorithmic form.

The algorithm which follows assumes that a context-free grammar *gram* and its alphabet *chars* are both given. The grammar *gram* is assumed to be a set of pairs <*int,seq*>, where *int* is an intermediate symbol, and *seq* is a tuple of characters corresponding to the right-hand side of a grammatical production. In what follows, two functions, *isgram(x)* and *isgraml(x,y)* are also used. The boolean-valued function *isgram(x)* has the value true if x is a subsequence of some sequence of characters constituting a valid sentence. The related Boolean function *isgraml(x,y)* tests a pair of sequences x,y and determines whether ...xy... can occur within a well-formed sentence within whose parse tree there exists no subtree to whose final twig the last character of x is attached. The routines *isgraml* and *isgram* make use of several auxiliary subroutines, one of which, *standptgram*, produces a grammar g', in the reduced form required for the above-described nodal span parsing algorithm; the sentences of the language described by g' are all the substrings of sentences of some initially given language.

```
definef mckeetable  (lmax, rmax, gram, root, chars, definite);
/*'driver' routine  for calculation of mckeeman tables */
/* ambig, oldambig, useambig, origins, startcovers, endcovers,
   covers, and table  are assumed to be global */
/* first calculate fixed tables useful below */
/* origins maps each right-hand side of a production
   into the corresponding left-hand side */
origins = {<x(2),x(1)>, x ∈ gram};
/* startcovers (resp. endcovers) maps each initial subsequence (resp.
   terminal subsequence) of the right-hand side of a production
   into the corresponding left-hand side */
startcovers = {<x(2)(1:k),x(1)>, x∈gram, 1≤k≤#x(2)};
endcovers = {<x(2)(k:),x(1)>, x∈gram, 1≤k≤#x(2)};
/* covers maps each subportion of the right-hand side of a
   production into the corresponding lefthand  side */
covers = {<x(2)(k:ℓ-k+1), x ∈ gram, 1 ≤ k ≤ #x(2), k≤ℓ≤#x(2)};
/* some initialization */ table = nℓ;
oldambig={<<c>,nult>, c ∈ chars};    ambig={<nult,<c>>,c∈chars};
/* generate sequence (1,1), (2,1), (1,2), (2,2), (3,2), ... */
m = 0; n = 1;
(while m le lmax or n le rmax)
   if m gt n then
        m = m-1; n = n+1;  useambig = oldambig;
   else
        m = m+1;   useambig = ambig;
   end if;
   if m le lmax or n le rmax then
                    extendres(m,n);
   end if;
end while;
if ambig ne nℓ then
   print 'ambiguous cases remaining', ambig;
   definite = f;
else
   definite = t;;
return table;
end mckeetable;
```

```
define extendres(m,n);
   extends context one symbol, to the left or right  depending
   on the relative values of m and n */

/* ambig, oldambig, useambig, origins, endcovers, startcovers,
   covers, and table          are assumed to be global */
newambig = nl;
(∀ampair ∈useambig, c∈ chars)
   <seqa, seqb> = ampair;
   /* extend by 1 character to right or left */
   if n gt m then
        seqb = seqb + <c>;
   else
        seqa = <c> + seqa;
        if m eq n and <seqa,seqb(1:n-1)>n∈ oldambig then
                                   continue   ∀ampair;;
   end if n;
   poss = nl; /* set of possible precedence relationships */
   if ∃x ∈ endcovers{seqa}|isgram(<x>+seqb) then
        1 ∈ poss;;
   if 1 < ∃n ≤ #seqa, x ∈ origins{seqa(n:)}
        |isgram(seqa(1:n-1) + <x> + seqb) then
        1 ∈ poss;;
   if seqa evenwith seqb then 2 in poss;;
   if ∃x ∈ startcovers{seqb}|isgraml(seqa<x>) then
        3 in poss;;
   if 1 ≤ ∃n < #seqb, x ∈ origins(seqb(1:n))
        |isgraml(seqa,<x>+seqb(n+1:)) then
        3 in poss;;
/* now either set up a new table entry or classify the context
                                   <seqa, seqb> as ambiguous */
   if (#poss) eq 1 then
        table(seqa, seqb) = ∋poss;
   else
        <seqa,seqb> in newambig;
        table(seqa,seqb) = 0;
   end if;
```

```
end ∀ampair;
oldambig = ambig; ambig = newambig;  return;
end extendres;

definef seqa evenwith seqb;
/* auxiliary routine to test for 'equality' case of precedence */
/* origins, startcovers, endcovers, and covers
                                        are assumed to  be global*/
return  if covers{seqa + seqb} ne nℓ then t
        else if 1 ≤ ∃k < #seqb, x∈endcovers{seqa+seqb(1:k)}
             |isgram(<x>+ seqb(k+1:)) then t
        else if 1 < ∃k ≤ #seqa, x∈startcovers{seqa(k:) +seqb}
             |isgram(seqa(1:k-1) +<x>) then t
        else if 1 < ∃k ≤ #seqa, 1 ≤  ℓ < seqb,
             x∈origins{seqa(k:) + seqb(1:ℓ)}
             |isgram(seqa(1:k-1) +<x>+seqb(ℓ+1)) then t;
        else f;
end evenwith;
```

Next we give code for the *isgraml(x,y)* function used above. As noted, this tests a pair of strings x,y and determines whether ...xy... can occur within a sentence, valid according to the grammar *gram*, within whose parse tree there exists no subtree to whose last twig the final character of x is attached. This test is made as follows. If the final character c of x is not the last character of any production, then *isgraml(x,y)* has the same value as *isgram(x+y)*. In the contrary case, the grammar *gram* is replaced by a modified grammar g'_c, and *isgraml(x,y)* is equal to *testgram(x'+y, g'_c, root, types)*. Here x' is identical with x, except that the final c in x is replaced by a new symbol c'; the Boolean fnnction *testgram(u,g',r,t)* returns the value t if the sequence u is part of a sequence grammatical according to the grammar g'; r and t are auxiliary parameters, explained below. The grammar g'_c is obtained by putting a third grammar g''_c into the special 'reduced' form

required for the nodal span parsing algorithm depicted earlier the present section. The symbols of g''_c are all the symbols of *gram*, plus the additional symbol c". Each production in *gram* is also a production of g''_c ; moreover, if p is a production of *gram*, and if c is replaced by c", in one occurrence of c on the right-hand side of p (but not an occurrence as the last character of the right-hand side of p), the production which results belongs to g''_c. Using the grammar g''_c and its reduced form g'_c , we can express the function *isgraml* very simply. Details follow below. Note that all necessary grammars g'_c are constructed in an initialization section of *isgram* . This construction uses a function *standptgram*, whose algorithm will be shown later, to obtain each g'_c from the corresponding g''_c. Note also that in the algorithms shown below we assume that grammars are given as sets of ordered pairs *<int,seq>*, where *int* is an intermediate symbol occurring on the left-hand side of a production p, and *seq* is the sequence of characters occurring on the right of p.

```
definef isgraml(x,y);
/* gram and root are shared with the routine mckeetable */
include mckeetable(gram,root);
/* initialize: build the collection of all symbols occurring
   as the final character of the right-hand side of some
   production; for each such character, form a 'primed' character
   and a modified grammar */
initially
   termsyms = {x(2)(#x(2)), x ∈ gram};
   primemap = {<c, newat>, c∈termsyms};
   chargrams = nl;
   (∀c ∈ termsyms)
        cprime = primemap(c);
        modgram = gram;
        (∀p ∈ gram, 1 ≤ k < #p(2) | p(2)(k) eq c)
             q = p; q(2)(k)  =    cprime;
             q in modgram;
        end ∀p;
```

```
        <c,modgram> in chargrams;
    end ∀c;
    chargraminf = {<c, standptgram(chargrams(c),root)>, c∈termsyms};
    /* the routine standptgram takes an arbitrary context free
         grammar g and its root r as arguments, and produces
         a related grammar g' whose sentences are the
         same as those of the language defined by g.
         moreover, g' is in the 'reduced' form assumed by the
         nodal span algorithms described in earlier paragraphs
         of the present section. the precise object returned
         by standptgram is a triple <g',r',ctypes>, where
         g' is as described, r' is the root of g', and
         ctypes is a mapping which sends each terminal symbol
         b of g' into the collection of all intermediate
         symbols A of g' for which there exists a production A→b */
end initially;
/* now we show the processing of a pair of character sequences x,y*/
c = x(#x);
if primemap(c) is cprime eq Ω then
    return isgram(x+y);
end if;
/* else */ <cgram, croot, ctypes> = chargraminf(c);
return  testgram(x(#x-1) + <cprime> + y, cgram, croot, ctypes);
end isgraml;
```

The routine *testgram* called in the semi-final line of the above procedure may be expressed very simply by using the nodal span parse procedure *nodeparse:*

```
definef testgram(string,gram,root,types);
nodeparse(string,gram,root,types,spans,divlis,amb);
return spans ne nℓ;
end testgram;
```

The routine *isgram(x)* used in the *mckeetable* routine is nsiderably simpler than, but similar to, the *isgraml* function. Here are the details of *isgram:*

```
definef isgram(x);
/*  gram and root are shared with the routine mckeetable */
include mckeetable(gram,root);
/* initialize by calculating the 'reduced' form of gram */
initially
   <standgram,standroot,types> = standptgram(gram,root);
   /* see the routine isgraml for a comment on standptgram */
end initially;
return testgram(x,standgram,standroot,types);
end isgram;
```

Next we discuss the manner in which a general context-free grammar of the form allowed in the preceding algorithm can be transformed into one having the more restrictive 'reduced' form considered in connection with the nodal span parsing method (cf. pp. 159-161). Observe that in the following discussion we consider two grammars as equivalent if their alphabets contain the same terminal symbols and if they generate the same set of valid sentences, i.e., the same non-empty valid strings containing terminal symbols only. Suppose then that a grammar is given as a set of products of the general form

(1) $$B \rightarrow AcdE...fG$$

We will initially allow products having null right-hand sides (these might be called 'erasing' productions); our first aim is to show how a grammar Γ containing such productions can be transformed into one which contains no such productions. This transformation is easy enough, and can be accomplished as follows.

i. Call an intermediate symbol B *erasable* if the null-string can be produced from it by some sequence of productions. Note that B is erasable if either B is the left-hand side of some 'erasing' production

$$B \rightarrow$$

or if there exists a production

$$B \rightarrow CDE...F$$

such that every symbol occurring on the right-hand side is known to be erasable.

ii. Drop all erasing productions from Γ. At the same time, given any production (1) of Γ, introduce as additional productions of Γ all those productions which result from (1) by dropping some set of erasable symbols from its right-hand side, but in such a way that at least one symbol remains on the right-hand side.

The set of productions obtained in this way constitutes a grammar, equivalent to Γ, but containing no erasing productions.

Assuming that a grammar is given as a set of pairs <*int*,*seq*> where *seq* is a tuple of characters, we can express this simple procedure in SETL as follows.

```
erasables = {hd x, x ∈ gram | x(2) eq nult};
(while ∃x ∈ gram|n hd x ∈ erasables and(∀c(n)∈x(2)|c∈erasables))
   hd x in erasables;
end while;
newgram = nl;
(∀x∈gram)
   <symb,seq> = x;
   eraselocs = {n, c(n) ∈ seq | c ∈ erasables};
   (∀spotset ∈ pow(eraselocs))
        newseq = [+: c(m) ∈ seq|m n ∈ spotset]<c>;
/* now have built up new right-hand side */
        if newseq ne Ω then  <symb, newseq> in newgram;;
   end ∀spotset;
end ∀x;
/* replace old grammar by new */  gram = newgram;
```

Next we aim to eliminate all single-character productions

(2) $$A \to B$$

of one intermediate symbol from another, and, at the same time, to eliminate terminal symbols from the right-hand side of every production except those having the single-symbol form

$$A \to b \ .$$

This again is an easy transformation, and can be accomplished as follows:

iii. For each terminal symbol a, introduce a new intermediate symbol $\tilde{A}$. Replace each occurrence of a on the right-hand side of a production of Γ by an occurence of $\tilde{A}$. Then add the productions

(3) $$\tilde{A} \to a$$

to Γ.

iv. Call A a *prime ancestor* of B if the single symbol B can be produced from A by some sequence of productions of the grammar Γ. Drop all single-intermediate-symbol productions (2) from Γ. At the same time, wherever there exists a production

$$B \rightarrow CDE...$$

with more than one symbol on its right-hand side, introduce a new production

$$A \rightarrow CDE... \quad .$$

In SETL, this procedure may be written as follows:

```
/* calculate the set of all symbols, and the set of all
   intermediate symbols of the grammar */
inter = hd[gram];
chars = inter + {p(2)(j), p ∈ gram, 1 ≤ j ≤ #p(2)};
term =  chars - inter;
intof = nl;  newg = nl;
(∀t ∈ term) intof(t) = newat;;
(∀x ∈ gram)
   <symb, seq> = x;  if (#seq) gt 1 then
<symb, [+: s(n)∈seq]<if intof(s) is int ne Ω then int else s>>
   in newg;          else x in newg;;
end ∀x;
/* which accomplishes step iii of the text */
singles = {<hd x(2); hd x>, x ∈ newg | #x(2) eq 1};
/* determine 'prime ancestor' relationship by transitive closure,
   cf. p. 119 */
pransc = closef(singles,inter);
newgram = nl;
(∀x ∈ newg | #x(2) gt 1 or hd x(2) ∈ term)
   <symb,seq> = x;  x in newgram;
   (∀y ∈ pransc(symb))<y, seq> in newgram;;
end ∀x;
/* now add terminal single-symbol productions to the new grammar,
   and set  gram  to new value */
gram = newgram + {<intof(t), <t>>, t ∈ term};
```

As transformed thus far, the grammar *gram* contains productions two types; productions A → a, and productions A → BCD...EF, where at least two symbols appear on the right, and where all symbols appearing on the right are non-terminal. For every production of the second kind whose right-hand side contains more than two symbols, we introduce a sequence of auxiliary intermediate symbols G, G',..., and replace the production A → BCD...EF by a set of productions

$$A \rightarrow BG$$

$$G \rightarrow CG'$$

$$G' \rightarrow DG''$$

$$\ldots$$

$$G''' \rightarrow EF$$

This step, which essentially completes the transformation of our grammar into the desired form, may be written in SETL as follows.

```
newgram = {x ∈ gram | (#x(2)) le 2};
(∀x ∈ gram | (#x(2)) gt 2)
   <left,seq> = x;
   (1 ≤ ∀n ≤ #seq-2)
        aux = newat;
        <left,<seq(n),aux>> in newgram;
        left = aux;
   end ∀n;
   <left, seq(#seq-1:)> in  newgram;
end ∀x;
gram = newgram;
```

A grammar in the precise form required by the **no**dal span parsing algorithm is then obtained from a grammar in the reduced form we have attained by making the following trivial final transformation.

```
finalgram = {<(x(2))(1), (x(2))(2), hd x>, x∈gram|(#x(2)) eq 2};
synt = {<(x(2))(1), hd x>, x ∈ gram | (#x(2)) eq 1};
syntypes = {<x, synt{x}>, x ∈ hd [synt]};
```

By combining of the code fragments appearing on the last few pages we obtain a routine *standgram(gram,root)* which takes an arbitrary context-free grammar as input, and as output returns a pair *<finalgram,syntypes>*; here *finalgram* is a grammar in 'reduced' form equivalent to the originally given grammar *gram*, and *syntypes* maps each terminal symbol b into the set of all intermediate symbols $\tilde{A}$ which can produce b via some production (3).

The transformations described in the last few pages enable us to represent the *standptgram* function required by the McKeeman table algorithm given above. The problem addressed in programming this function is as follows: given a string S of tokens, we wish to determine if S is a *part* of any sentence grammatical according to a given grammar Γ. To solve this problem, we extend the grammar Γ, obtaining a larger grammar Γ' which generates all strings S which are parts of strings generated by Γ. This is done in the following way. For each intermediate symbol A of Γ, introduce three additional symbols, which we call A_ℓ, A_r, and $A_{\ell r}$; if a is a terminal symbol, a_ℓ, a_r, and $a_{\ell r}$ will all be equal to a. For each production

(4) $$A \to BCD...EFG$$

of Γ, introduce new productions

$$A_\ell \to B_\ell CD \ldots EFG$$
$$A_\ell \to C_\ell D \ldots EFG$$
$$A_\ell \to D_\ell \ldots EFG \text{ , etc.}$$

by removing zero or more symbols from the left-hand end of the right side of (4). Similarly, introduce productions

$$A_r \rightarrow BCD...EF_r$$

$$A_r \rightarrow BCD...E_r \quad , \text{ etc.}$$

by removing zero or more symbols from the right-hand end of the right side of (4). Moreover, introduce productions

$$A_{\ell r} \rightarrow C_\ell D...E_r \quad , \text{ etc.}$$

by removing zero or more symbols both from the right and from the left-hand end of the right of (4). For each symbol D which occurs on the right-hand side of a production (4), introduce a production

$$A_{\ell r} \rightarrow D_{\ell r} \quad .$$

Next, take the root symbol $\bar{A}$ of Γ, introduce a new root symbol $\hat{A}$ for Γ', and introduce three productions,

$$\hat{A} \rightarrow \bar{A}_\ell$$

$$\hat{A} \rightarrow \bar{A}_r$$

$$\hat{A} \rightarrow \bar{A}_{\ell r} \ .$$

Finally, introduce productions

$$A_\ell \rightarrow A$$

$$A_r \rightarrow A$$

for each intermediate symbol A. The grammar Γ' we require consists of all these productions, and has $\hat{A}$ as its root symbol; we leave it to the reader to show that Γ' generates precisely those sentence-forms which are parts of sentence-forms generated by Γ.

Assuming that *root* designates the root-symbol of Γ, a procedure *partgram* which constructs Γ' from Γ may be depicted in SETL as follows.

```
definef partgram(gram,root);
inter = hd[gram];  ℓ = nℓ;  r = nℓ;  ℓr = nℓ;
(∀x ∈ inter) ℓ(x) = newat;  r(x) = newat; ℓr(x) = newat;;
newroot = newat; extgram = gram;
/* initialize new grammar */
(∀x ∈ inter)
   <ℓ(x),<x>> in extgram;
   <r(x), <x>> in extgram;
   (∀seq ∈ gram{x}, 1 ≤ n ≤ #seq)
        <ℓr(x),ℓr(seq(n))> in extgram;
   <r(x), [+: 1≤m≤n] <if m lt n then seq(n) else r(seq(n))>
        in extgram;
   <ℓ(x), [+: n ≤ m ≤ #seq]<if m gt n then seq(m) else ℓ(seq(n))>
        in extgram;
        (1 ≤ ∀nn < n)
   <ℓr(x), [+: nn ≤ m ≤ n] <if m eq nn then ℓ(seq(m))
   else if m eq n then r(seq(m)) else seq(m)> in extgram;
        end ∀nn;
   end ∀seq;
end ∀x;
extgram = extgram with <newroot,<ℓ(root)>> with<newroot,<r(root)>>
                  with <newroot,<ℓr(root)>;
return <extgram,newroot>;
end partgram;
```

By combining the routine *standgram* described earlier with the routine *partgram* which has just been given, we obtain the routine *standptgram* required for the *isgraml* subroutine of the McKeeman procedure. The detailed definition of *standptgram* is as follows:

```
definef standptgram(gram,root);
<newgram,newroot> = partgram(gram,root);
<finalgram,syntypes> = standgram(newgram,newroot);
return <finalgram,newroot,syntypes>;
end standptgram;
```

We now give a few additional algorithms which arise in connection with compilers. The first may be used in processing the EQUIVALENCE declarations which occur in the FORTRAN language. In this language, one is allowed to specify an overlay pattern for arrays by stating that location k in an array A is to be the same as location k' in another array A'; then the j-th address in A' is the address j+k-k' in A. The algorithm (due to Fisher and Galler) that we give below takes a set of declarations of this sort and divides the set of all arrays mentioned in them into disjoint families, within each one of which every array is related by a given offset to a specified *master array*. In the process, the whole set of declarations is checked for consistency.

We write the algorithm as a function, whose input is a set of *declaration triples* <*arraya, arrayb, offset*> , *offset* being an integer quantity. A flag is set if analysis of the whole set of triples reveals an inconsistency. The map *master* gives, for each array A, a pair <*arrayb,offset*> consisting of an array to which A is related by a definite offset, and also gives this offset.

```
definef equivproc(dectrips,err);
err = f;  master = nl;  arrays = nl;
(∀trip ∈ dectrips)
   <arra,arrb,offs> = trip;
   arra in arrays; arrb in arrays;
   /* chain to ultimate masters */
   (while master(arra) ne Ω)  /* correcting offsets */
        <arra,offs>=<hd master(arra),offs+master(arra)(2)>;
   end while;
   (while master(arrb) ne Ω)
        <arrb,offs>=<hd master(arrb), offs-master(arrb)(2)>;
   end while;
/* offs is now the offset relating two formerly independent
   master arrays.  check for contradiction */
   if arra eq arrb and offs ne 0 then err = t;  return Ω;;
   /* otherwise make second array master of first */
```

```
   master(arra) = <arrb,offs>;
end ∀trip;
/* now summarize all the relationships that have been collected*/
(∀arra ∈ arrays)
   <arr,offs> = <arra,0>;
   (while master(arr)ne Ω)
        <arr,offs> =<hd master(arr), offs+master(arr)(2)>;
   end while;
   master(arra) = <arr,offs>;
end ∀arra;
return master;
end equivproc;
```

Next we shall describe, in somewhat simplified form, the structure of a macroprocessor, not quite as powerful as that specified for SETL, but nevertheless providing quite a useful degree of function. The facilities supported by the macroprocessor to be described are as follows.

a. Macro definitions; macros with and without arguments.

In the macro-system to be described, macros with parameters are declared in the form

(1) define macro *macroname* (arg_1, ...,arg_n); *body* endm;

Macros without arguments are declared in the form

(2) define macro *macroname*; *body* endm;

We suppose that input to the macroprocessor comes from a lexical scanner of the kind described in Section 2 of the present item, and that the macroprocessor is in turn required to supply tokens to some other routine forming part of an overall language processing system.

In (1) and (2), *macroname* is a token which becomes the name of a macro; the arguments arg_1, ..., arg_n are also tokens, which must be distinct, and which become the arguments of this macro. The *body* occurring in (1) and (2) is a text, which is decomposed into tokens.

A macro defined by (1) and (2) may be invoked at any point subsequent to its definition by writing

(3) *macroname*($argum_1$, ..., $argum_n$)

or, if there are no arguments, simply by writing

(4) *macroname* .

The number of arguments in (1) and (3) must match, though null arguments are permitted in (3). Each argument $argum_j$ may be any sequence of symbols which is parenthesis-balanced and contains no exposed commas, i.e., no commas not enclosed in parentheses. Moreover, the token combinations

(5) define macro

and

(6) end;

are prohibited from appearing in a macro's body.

Macro invocations are expanded by substituting $argum_j$ for each occurrence of arg_j in the text *body* appearing in definition (1), and issuing the resulting stream of tokens instead of the stream initially input to the macroprocessor. If this stream is found to contain a macro invocation, this inner invocation is expanded, and so on recursively. Thus macro bodies may usefully contain macro invocations. It follows from the restrictions stated just above (cf. (5), (6)) that *direct* inclusion of macro definitions within macro bodies is excluded; however, the macroprocessor to be described does allow macro definitions to be included in macro bodies in a somewhat roundabout way, which we leave it to the reader to discover.

b. Generated symbols.

It is often useful to have each expansion of a macro generate symbols which are unique to that expansion. This, for example, allows statement labels to be used in macros without label-repetition conflicts arising. The macroprocessor to be described will support the following generated symbol facility. A macro which is to generate symbols should be defined in the form:

(7) define macro *macroname*$(arg_1,\ldots,arg_n/xarg_1,\ldots,xarg_k)$;*body* end;

if the macro involves n arguments and k generated symbols, or, only generated symbols but no arguments are wanted, in the form

(8) define macro *macroname*(/$xarg_1$,...,$xarg_k$); *body* end;

The invocation of a macro defined in the form (7) is identical with that of a macro defined in the manner (1); likewise the invocation of a macro defined in the form (8) is identical with that of a macro having a definition (2). However, the applicable macro-expansion rules are somewhat modified. The string $argum_j$ is still substituted for each occurrence of arg_j in the text *body* of definition (7). In addition, unique symbols $symb_1$, ..., $symb_k$ are generated, and $symb_j$ is substituted for each occurrence of $xarg_j$ in *body*. The generated symbols $symb_j$ consist of the prefix ZZZZ, followed by decimal digits; symbols of this special form should not be used explicitly for any other purpose.

c. Macro redefinition, macro drop.

Macros may be redefined. The special definition

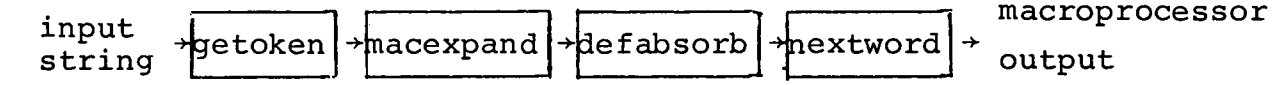
define macro *macroname* end;

removes *macroname* from macro status.

The macro facilities described above are all supported by a relatively simple processor, which we assume is to function as an interface between a parser and a lexical scanner. The parser calls upon the macro processor when it requires an additional token. The macro processor either supplies a token, produced as a result of macro expansion, or, if no macro expansion is appropriate, calls upon the lexical scanner for a new token to be passed along. The macroprocessor acts recursively, but not written as a recursive routine; instead, it secures

recursive effects by the manipulation of a pushdown stack. An internal "library" of previously declared macros is maintained by the macroprocessor, in the form of a function *macrodict* which maps each macro onto a triple <*nxargs*, *argfn*, *mbody*>. Here *nxargs* is the number k of generated arguments appearing in (7) or (8), *mbody* is the declared text of the macro, and *argfn* maps each prototype argument (whether supplied or generated) of the macro onto its serial number.

The overall structure of the macroprocessor is shown in the following diagram.



An innermost routine, called *getoken*, is essentially the same as the lexical scanner described in Section 2 of the present item. The next innermost routine is called *macexpand*. It receives tokens from *getoken*, and supplies tokens to a routine called *defabsorb*. The procedure *macexpand* works with a stack representing macros to be expanded. If this stack is **not** empty, *macexpand* supplies tokens by performing expansion operations as indicated by the topmost stack entries. Otherwise *macexpand* simply obtains a token from *getoken* and passes it along.

The routine *defabsorb* receives tokens from *macexpand* and passes them along to the outermost routine of the macroprocessor system, which is called *nextword*. *Defabsorb* monitors the token stream which it is handling for the special token combination which indicates the start of a new macro definition. When the start of a macro definition is found, *defabsorb* builds up a list of formal arguments of the macro, and enters the macro body into the macro dictionary function *macdict*. Note therefore that all text associated with macro definitions is completely absorbed by *defabsorb*.

The routine *nextword*, which is the outermost portion of the macroprocessor) gets tokens from *defabsorb* and passes them out to the language processing system of which the macroprocessor forms part. In addition, *nextword* monitors the token stream fo: tokens flagged as macro names. When a macro name is found, a

new layer of macro expansion is set up. To begin the expansion a macro, *nextword* collects the actual arguments of the macro from the token stream which it is handling.

If there are any generated arguments, i.e., if the macro to be expanded has been defined in the form (7) or (8), new symbols are generated and added to the tuple of actual arguments to be used by *macexpand*. A pointer to the initial symbol of the body of the macro to be expanded is also stacked. *Nextword* then initiates macro expansion simply by calling upon *defabsorb* to supply an additional token.

Three auxiliary routines are used. The subroutine *getargs(argfn)* is called by *defabsorb* to read a comma-separated string of macro arguments and to extend an argument to argument-number map as new arguments are read. Two routines *printerror(msg)* and *printwarn(msg)* are called, the first to record fatal errors and to print error messages and accompanying diagnostic text, the second to print warning messages.

Two main data objects are used by the macroprocessor. The first, *macdict*, maps each word which has been declared to be a macro into a tuple describing the arguments and text of the macro. The second, *expstack*, is a pushdown stack used to obtain recursive effects during macro processing.

The counter *xarggenctr* is used to ensure uniqueness of generated argument symbols.

The routine *nextword* is set up in a way which makes "backup" available to whatever parser calls on *nextword* for tokens. This is done as follows. Within *nextword*, a first-in-first-out queue of tokens called *reserve* is maintained. When called, *nextword* examines this tuple. If *reserve* is not null, its last component is detached and returned as the value of *nextword*. In the contrary case, *nextword* calls on *defabsorb* to supply a token. Therefore,

to return a symbol to *nextword* one simply appends it to *reserve*.

A similar but simpler mechanism, supporting a single word of backup, is provided within *macexpand*. The global word *macexpgiveback* (see next page) is used to hold a symbol returned to *macexpand* from *defabsorb*.

We assume in what follows that tokens are supplied to the innermost routine of the macroprocessor system, i.e., to the routine *macexpand*, by a lexical scanner like that described in Section 2 of the present item. As already stated, this routine is called *getoken* in the SETL text which follows. *Getoken* is assumed to return ordered pairs of the form <*string*,*type*>, in which *string* is a token-representing character string, and in which *type* is a numerical code defining the lexical type of string. We do not include code for *getoken* with the routines which now follow.

```
definef nextword;
/* macdict, expstack, and reserve are assumed to be global */
initially
     reserve = nult; xarggenctr = 0; macdict = nl;
     expstack = nult;
end initially;
if reserve ne nult then
     word = reserve(1); reserve = reserve(2:);
     return word;
end if;
/* otherwise get additional token from defabsorb */
getword: word = defabsorb( );
if macdict(word) is macinf eq Ω then /*word is not a macroname*/
     return word;;
/* else word is a macro name. determine number and nature of
     arguments; also, determine macro body text */
<nxargs,argfn,mbody> = macinf;
```

```
nargs = #argfn-nxargs;  /* number of nongenerated  arguments */
  allinf = <1,mbody,argfn>;  /* first fragment of macro-call
        information */
argtupl = nult; /* collection of arguments */
if nargs gt 0 then go to getargs;
/* otherwise merely generate additional arguments, as required */
genargs: (1 ≤ ∀j ≤ nxargs)
            argtupl = argtupl + <'zzzz' + dec xarggenctr>;
            xarggenctr = xarggenctr + 1;
         end ∀j;
         expstack(#expstack+1)=mcallinf+<argtupl>;
           /* completing set-up of macro call */
         go to getword;
getargs:/* the macro has arguments.
             collect its arguments out of the token stream */
         if (hd defabsorb( )) ne '(' then
            printerrror('missing macro arguments. macro
              expansion suppressed');
            return word;
         end if;
         (1 ≤ ∀j ≤ nargs)
            parencount = 0; /* unmatched parenthesis count */
            curargtupl = nult; /* argument being built up */
            (while hd(defabsorb( ) is word) ne ',' or
                                          parencount gt 0)
               if (hd word is token) eq er then
                  printerror ('improper end of record');
                  return word;
               else if token eq ')' then
                  parencount = parencount - 1;
                  if parencount eq -1 then quit ∀j;
               else if token eq '(' then
                  parencount = parencount + 1;
               end if (hd word;
               curargtupl(#curargtupl + 1) = word;
            end while;
            argtupl(#argtupl+1) = curargtupl;
```

```
        end ∀j;
        if #argtupl lt nargs then
             printerror('missing parameters in macro call');
             nxargs = nxargs + nargs - #argtupl;
        else if parencount ne -1  then
             printerror('surplus parameters ignored in macro call');

        end if;
        go to genargs;
        end nextword;

        defined defabsorb;
        /* routine to analyze and record macro definitions */
        /* nametype is assumed to be a globally known constant */
        /* macexpgiveback is assumed to be global */
scan:   token = hd (macexpand() is word);
        xtoken = hd (macexpand() is xword);
        if <token,xtoken> eq<'endm', ';'> then
             printerror('improper macro close before opening');
        else if <token,xtoken> ne <'define', 'macro'> then
        /* give back one word and return the other */
        macexpgiveback = xword;
        return word;
        end if;

         /*here we begin to read a macro definition */
        nargs = 0; argfn = nl; /* argument to argument - number map */
        type = (macexpand() is mname) (2);
        if type ne nametype then
             printerror ('name missing in macro definition. definition
               ignored');
             return word;
        end if;
```

```
   /* otherwise have valid macro name. look for following argument
        list, semicolon, or 'endm' */
   token = hd (macexpand() is word);
   if token eq '(' then
        go to getargums;
   else if token eq ';' then
        if macdict (word) ne Ω then
        printwarn ('prior macro definition is being changed');
        end if macdict;
        go to getbody;
   else if token ne 'endm' then
        printerror ('improper continuation of macro definition.
            definition ignored');
        return word;
   end if token;
   /* here we have seen define macro macroname endm.  look for
        semicolon */
   if hd( macexpand() is word) ne ';' then
        printerror ('improper termination of macro drop');
        macexpgiveback = word;
   end if;
   if macdict (mname) eq Ω then
        printwarn ('drop applied to non-macro name');
   else
        macdict (mname) = Ω;
   end if;
   go to scan;
getargums:     /* scan for the arguments of a macro */
   if  hd(macexpand( ) is word) eq '/' then
        go to getxargs; /*since there are no true arguments */
   else
        macexpgiveback = word; /*send word back to macexpand */
   end if;
   getargs(argfn); /* get chain of arguments separated by commas*/
   token = hd (macexpand( ) is word);
   if token eq '/' then go to getxargs;;
   nargs = #argfn; /* record number of true macro arguments */
```

```
testalistend: if token ne ')' then
              printerror ('illegal termination of macro argument lis
                   definition ignored');
              return word;
         end if;
         /* now check for ';' following argument list */
         if hd (macexpand() is word) ne ';' then
              printerror ('illegal termination of macro argument list');
              macexpgiveback = word;
         end if;
         /* at this point the macro argument list is complete, and we
              begin to collect the body of the macro */
getbody:mbody = nult;
loop:    token = hd (macexpand() is word);
         xtoken = hd (macexpand() is xword);
         if <token,xtoken> eq<'endm',';'> then /* close the definition */
              macdict(mname) = <#argfn-nargs,argfn,mbody>;
              go to scan;
         else if <token,xtoken> eq <'define', 'macro'> then
              printerror ('improper macro definition within macro body');
         else if token eq er then
              printerror ('end of file encountered in macro definition');
              return word;
         end if;
         /* otherwise merely add token to mbody and continue */
         mbody (#mbody+1) = token;
         macexpgiveback = xword;  /* send xtoken  back to macexpand */
         go to loop;
getxargs:/*here we build up the list of extra arguments of a macro */
         nargs = #argfn; /* record number of ordinary macro arguments */
         getargs (argfn);
         token = hd (macexpand() is word);
         go to testalistend;
         end defabsorb;
```

```
      define getargs (argfn);
      /* auxiliary procedure to build up list of arguments */
      /* nametype is assumed to be a globally known constant */
   getaloop:if(macexpand( ) is word)(2) ne nametype then
            printerror ('missing argument name in macro argument
            return;                                          list');
      else if argfn(word) ne Ω then
            printerror ('duplicate argument name in argument list.
                 duplicate ignored');
      else
            argfn(word) = #argfn + 1;
      end if;
      if hd (macexpand() is word) eq ',' then go to getaloop;
      macexpgiveback = word;  /* send word  back to macexpand */
      return;
      end;

      definef macexpand;
      /* routine to expand macros */
      /* macexpgiveback and expstack are assumed to be global */
      if macexpgiveback ne Ω then
            keep = macexpgiveback; macexpgiveback = Ω;
            return keep;
      end if;
start: if expstack eq nult then return gettoken( );;
      /* otherwise we are in process of expanding a macro */
expand: exptop = expstack(#expstack);
      <symbno,body,argfn> = exptop(1:3);
      if symbno gt #body then /* finished with expansion */
         expstack(#expstack) = Ω; go to start;
      end if;
      symbol = body(symbno);
      if argfn(symbol) is argno eq Ω then
                  /* symbol is not argument */
         exptop(1) = symbno+1;
         expstack(#expstack) = exptop;
```

```
   return symbol;
end if;
/* else symbol is argument; a new level must be built on the
        expansion stack */
argtupl = exptop(4); /* tuple recording actual macro arguments */
expstack(#expstack+1) = <1,nl, argtupl(argno), nult>;
/* this makes top of stack appear to be a macro whose expansion is

   the text of the argument actually supplied */
go to expand;
end macexpand;
```

8 Algorithms for theorem-proving by the resolution method.

If algorithms are to discover and perform many of the powerful simplifying transformations utilized by human programmers, they will have to be able to find the proofs of correctness which programmers use to justify such transformations. This is but one of the many considerations which lend interest to algorithms which find proofs of mathematical theorems. Such algorithms, some of which we shall present in the present section, generally fall into one of two rather different classes. One type of algorithm is designed specifically for application in a given field of mathematics (such as Euclidean plane geometry), and incorporates structures and procedures particular to this field. The other type of theorem-prover, which is the type with which we shall now concern ourselves, uses general methods belonging to mathematical logic, and given axioms describing any field can in principle prove theorems in that field. (However, such algorithms may be inefficient, and may require entirely impractical amounts of time and memory.)

Any algorithmic theorem-proving scheme must of course rest upon a formalization of the part of mathematics which it is to cover, i.e., upon some method for representing a mathematical subject domain by a set of formulae subject to algorithmic manipulation. To this end, the algorithms which we shall consider make use of the so-called *first order functional calculus* of mathematical logic. In this calculus, formulae are built up from the following elements:

i. The basic *boolean connectives: and, or, not, implies, is equivalent to;* or, as logicians normally write them:

$$\& \ , \ \vee \ , \ \sim \ , \ \supset \ , \ \equiv$$

ii. *Variables*, representing objects varying over a given mathematical domain. Normally written x, y, etc.

iii. *Constants*, representing specific objects. Normally written c, d, etc.

iv. *Function symbols*, representing particular basic or derived mappings of objects to objects within a mathematical

domain, and normally written f, g, etc. Each function symbol is understood to have some fixed number of arguments. By supplying arguments to a function symbol, we can form symbolic structures representing constant or variable objects, as in the following examples:

f(x); f(c); g(x,y); g(c,f(x)); g(c,f(f(g(f(y),g(y,d))))) .

v. *Predicate* symbols, representing particular functions whose values are all either $\underline{t}$ or $\underline{f}$. Normally written P, Q, etc. Supplying a predicate with its fixed number of arguments, we form *units* representing constant or variable boolean values, e.g.

P(f(x)); Q(c,c); Q(g(x,y),f(x)); Q(g(c,y),g(y,f(c))) .

vi. The boolean quantifiers $(\forall x)$, $(\exists x)$, where x denotes any variable. By combining units with connectives and quantifiers in syntactically valid ways, we form all the allowed formulae of the first order functional calculus. Examples of such formulae are

$(\forall x)f(x) \supset f(c)$; $g(c,c) \supset (\exists x)(\exists y)(g(x,y) \vee \sim f(x))$, etc.

Formulae of the first order calculus can be used to represent the assertions of any mathematical domain. For example, to represent the commutative and distributive laws of algebra, we introduce function symbols called *plus* and *times*, and a predicate symbol called *equals*, and write the formulae

(1) $(\forall x)(\forall y)$ equals (plus(x,y), plus(y,x))
$(\forall x)(\forall y)$ equals (times(x,y), times(y,x))
$(\forall x)(\forall y)(\forall z)$ equals (plus(times(x,y),times(x,z)),
times(x,plus(y,z)))

To represent the algebraic fact that there exists a zero element which is an additive unity, we introduce a constant symbol *0* and write

(2) $(\forall x)$ equals (plus(x,0),x);

to represent the existence of an additive inverse, we write

(3) $(\forall x)(\exists y)$ equals (plus(x,y),0) .

A mathematical structure described by one or more formulae C

the first order functional calculus defines an *interpretation* of the formulae which describe it. More specifically, consider a collection S of formulae C of the first order functional calculus. In the formulae C, certain constant symbols, function symbols, and predicate symbols will occur. An interpretation of S is given by specifying:

(i) a nonempty set U called the *universe* of the interpretation;

(ii) for each constant symbol occurring in a formula C, some definite object of U;

(iii) for each function symbol of n parameters occurring in a formula C, some mapping of the n-fold cartesian product U×...×U into U;

(iv) For each predicate symbol of n parameters occurring in a formula C, some mapping of the n-fold cartesian product U×...×U into the two-element set {t,f}.

Given such an interpretation, each formula of C becomes a definite mathematical proposition, having one of the values 'true', 'false'. For example, we may consider the two following interpretations of the formulae (1), (2), (3) shown above:

Interpretation A. *U* is the set of all real-valued functions of a real variable; *plus* designates addition of real valued functions; *times* designates multiplication of real valued functions; the predicate symbol *equals* designates function equality; the constant symbol 0 designates the function all of whose values are zero.

Interpretation B. *U* is the set of all non-negative integers, *plus* designates integer addition, *times* integer multiplication; the predicate symbol *equals* designates equality of integers, the constant symbol 0 designates the zero integer.

In both these interpretations, formulas (1) and (2) take on the value true. However, formula (3) takes on the value true in interpretation A, but the value false in interpretation B.

An interpretation which makes every one of the statements of a family S take on the value <u>true</u> is called a *model* of S. For example, consider the following set S_1 of statements, in which the function symbols *i* and *prod*, the constant symbol *e*, and the predicate symbol *equals* appear.

(4)
$$(\forall x)\ equals\ (prod(e,x),x)$$
$$(\forall x)\ equals\ (prod(i(x),x),e)$$
$$(\forall x)(\forall y)(\forall z)\ equals\ (prod(x,prod(y,z)),prod(prod(x,y),z))$$
$$(\forall x)\ equals\ (x,x)$$
$$(\forall x)(\forall y)(equals(x,y) \supset equals(y,x))$$
$$(\forall x)(\forall y)(\forall z)((equals(x,y)\ \&\ equals(y,z)) \supset equals(x,z))$$
$$(\forall x)(\forall y)\ (equals\ (x,y) \supset equals(i(x),i(y)))$$
$$(\forall x)(\forall y)(\forall z)(\forall u))((equals(x,y)\ \&\ equals(z,u) \supset equals(prod(x,z),\ prod(y,u)))$$

What is ordinarily called a *group* is simply a model of the statements (4) in which "=" is interpreted as equality. The interpretations of *e*, *prod*, and *i* are then the group identity element, group multiplication, and the function which maps each group element into its inverse, respectively. In fact the statements (4) are a formal system of *axioms for group theory.*

Next, suppose we wish to show that some particular statement *T* can be proved using a *finite* set of axioms $A_1,\ldots,A_m$. We take this to mean that we want to show that:

Every model of $A_1,\ldots,A_m$ is also a model of T.

For example, to show that the axioms S_1 listed above imply the elementary group-theoretic theorem

(5)
$$(\forall x)\ equals(prod(x,i(x)),e)$$

is to establish that (5) is true in every model of the statements (4). This point of view towards provability from axioms is usually characterized as *semantic* or *model-theoretic.* Textbooks of logic tend to emphasize a different, *syntactic* point of view in which to derive a T from $A_1,\ldots,A_m$ is to use a certain set of formal *rules of inference* to obtain T from $A_1,\ldots,A_m$. However, these rules of

ference (the inference rules of first order functional calculus) may be shown to have the following property:

T can be obtained from $A_1,\ldots,A_m$ *by employing the rules of inference of first order functional calculus if and only if every model of* $A_1,\ldots,A_m$ *is also a model of T.*

This is the content of the fundamental *completeness theorem of Gödel*, which therefore guarantees that the syntactic and the semantic notion of provability are equivalent.

Every interpretation of T must make T either true or false. Hence, every model of $A_1,\ldots,A_m$ is either a model of T or of $\sim$T. Thus, to say that every model of $A_1,\ldots,A_m$ is also a model of T is equivalent to saying that *no model of* $A_1,\ldots,A_m$ *is also a model of* $\sim$T, or equivalently that $A_1,\ldots,A_m$, $\sim$T have no model. All the proof procedures to be discussed in this section use this approach, i.e., attempt to prove that $A_1,\ldots,A_m$ imply T by showing that $A_1,\ldots,A_m$, $\sim$T have no model.

Faced with this latter task, we always begin by subjecting the statements $A_1,\ldots,A_m$, $\sim$T to certain preliminary transformations, which transform them into new statements $C_1,\ldots,C_k$ which have a model if and only if $A_1,\ldots,A_m$, $\sim$T have a model. These transformational steps are as follows:

Step 1. *Relabel variables.* If the same variable occurs in more than one quantifier in the same statement, use a new variable for one of them. For example, $(\forall x)R(x) \vee (\forall x)S(x)$ should be written as, say, $(\forall x)R(x) \vee (\forall y)S(y)$.

Step 2. *Eliminate* $\supset$ *and* $\equiv$. Whenever $\supset$ and $\equiv$ occur, make the following replacements:

Replace $A \rightarrow B$ by $(\sim A) \vee B$.
Replace $A \equiv B$ by $(A \,\&\, B) \vee [(\sim A) \,\&\, (\sim B)]$.

Step 3. *Move* $\sim$ *inwards.* Where possible make the replacements:

Replace $\sim(x)M$ by $(\exists x) \sim M$.
Replace $\sim(\exists x)M$ by $(x) \sim M$.

Replace $\sim(M \;\&\; N)$ by $(\sim M) \vee (\sim N)$.

Replace $\sim(M \vee N)$ by $(\sim M) \;\&\; (\sim N)$.

Replace $\sim\sim M$ by M .

Ultimately the statements are obtained in a form where each $\sim$ occurs immediately preceding an atomic formula.

For atomic formulas we write $\bar{R}(U_1,\ldots,U_m)$ instead of $\sim R(U_1,\ldots,U_m)$; $\bar{R}(U_1,\ldots,U_m)$ and $R(U_1,\ldots,U_m)$ are both referred to as literals.

Thus beginning with

$$(\forall x)\{R(x) \vee \sim(\exists y)[S(x,y) \;\&\; (R(x) \vee U(x))]\}$$

we obtain

$$(\forall x)\{R(x) \vee (\forall y)[\sim S(x,y) \vee (\sim R(x) \;\&\; (\sim U(x))]\}$$

Step 4. *Eliminate existential quantifiers.* Cross out each existential quantifier, say $(\exists y)$. The corresponding variable (in this case y) is replaced by $g(x_1,\ldots,x_m)$, where g is a new function symbol, and $x_1,\ldots,x_m$ are all of the variables occurring in universal quantifiers to the left of the existential quantifier (in this case $(\exists y)$).

In case there are no universal quantifiers to the left of the existential quantifier, the variable is replaced by a new constant symbol g; this case may be included in the general case *by regarding a constant symbol as just a function symbol of* 0 *arguments.*

To see that step 4 preserves the properties of having or not having a model, we note that if the statements had a model before one use of step 4, for each $x_1,\ldots,x_m$ of the universe there would be one or more values of y making the statement being processed true. Hence we may take as the interpretation of g a function such that for each $x_1,\ldots,x_m$ the statement is true when $y = g(x_1,\ldots,x_m)$. Conversely, if the statements have a model after processing, then $y = g(x_1,\ldots,x_m)$ will make the statement true for all $x_1,\ldots,x_m$ so that the original existential condition likewise has a model.

Step 5. *Advance universal quantifiers.* Move all universal quantifiers to the left so that the statement has the form of a sequence of universal quantifiers followed by a quantifier-free expression.

Step 6. *Distribute, wherever possible,* & *over* ∨. I.e. replace & B) ∨ C by (A ∨ C) & (B ∨ C).

After steps 1-6 have been performed each statement must have the form

$$(\forall x_1)(\forall x_2)\ldots(\forall x_n)\ [C_1 \ \& \ C_2 \ \& \ \ldots \ \& \ C_m] \ ,$$

where, for each C_i

$$C_i = \ell_1^{(i)} \vee \ell_2^{(i)} \vee \ldots \vee \ell_{r_i}^{(i)} \ ,$$

and where each $\ell_j^{(i)}$ is either a unit or the negative of a unit. In this expression each C_i is called a *clause*.

Step 7. *Simplify.* If a unit and its negation both occur in a clause C_i, strike out the entire clause. If a literal occurs twice as a literal of the same clause C_i, strike out one occurrence. (E.g. the clause $\ell_1 \vee \ell_2 \vee \ell_1$ is to be replaced by $\ell_1 \vee \ell_2$.)

Step 8. *Abbreviate,* by omitting the universal quantifiers prefixed to each remaining statement; and break up each disjunction $C_1 \ \& \ \ldots \ \& \ C_k$ into its separate clauses $C_1, \ldots, C_m$.

This completes our preliminary processing of an initially given set of statements $A_1,\ldots,A_n, \sim T$. Note that it is not claimed that the set of statements $C_1,\ldots,C_m$ which results from the processing is equivalent to $A_1,\ldots,A_n,\sim T$. Rather, it is claimed that the set $A_1,\ldots,A_n,\sim T$ has a model if and only if the set $C_1,\ldots,C_m$ has a model. The set of statements $C_1,\ldots,C_m$ is said to be the Herbrand normal form of the initial set $A_1,\ldots,A_n,\sim T$.

The *Herbrand universe* (HU) of $C_1,\ldots,C_m$ is defined as follows: let $K = \{k_i\}$ be the set of all constant symbols occurring in $C_1,\ldots,C_m$, and let $\{f_j\}$ be the set of all function symbols occurring in $C_1,\ldots,C_m$. (If no constant symbols occur in $C_1,\ldots,C_m$, we invent one nominal constant k_0, and let $K = \{k_0\}$.) Then the elements of HU are all the symbols k_i, as well as all the formulae $f(e_1,\ldots,e_p)$, where f is a function symbol of p mal parameters, and $e_1,\ldots,e_p$ are themselves symbols of HU.

It is plain that each function symbol f of p formal parameters automatically defines a p-parameter mapping of HU into itself, namely the mapping which takes p formulae $e_1,\ldots,e_p$ of HU into the formula $f(e_1,\ldots,e_m)$ of HU. Note that if $C_1,\ldots,C_m$ have a model with universe U, then each element k of K corresponds to some element μu of U, and each function symbol f of p formal parameters to a p-parameter mapping of μf of U into itself. Thus the recursive rule

$$\mu(f(e_1,\ldots,e_p)) = (\mu f)(\mu e_1,\ldots,\mu e_p)$$

defines a mapping of HU into U.

A model of $C_1,\ldots,C_m$ associates a p-parameter, boolean valued function μP defined on U with each predicate symbol P of p parameters, and we can carry this over to HU by insisting that

$$(\mu P)(e_1,\ldots,e_p) = (\mu P)(\mu e_1,\ldots,\mu e_p)$$

for each sequence $e_1,\ldots,e_p$ of elements of HU. This observation makes it plain that $C_1,\ldots,C_m$ has a model (with arbitrary universe) if and only if it has a model with universe HU. Note now that finding a model of $C_1,\ldots,C_m$ with universe HU is precisely the same as finding some way of assigning a truth value to each unit generated using the predicate symbols, function symbols, and constant symbols of $C_1,\ldots,C_m$, in such a way as to make every substituted instance of each of the formulae $C_1,\ldots,C_m$ have the value true. Hence this will be impossible, i.e., $C_1,\ldots,C_m$ (or equivalently $A_1,\ldots,A_m,\sim T$) will have no model, if and only if a contradiction can be derived from $C_1,\ldots,C_m$ by some finite process of substitution and boolean calculation. This is Herbrand's theorem, which we restate for emphasis as follows:

Let $A_1,\ldots,A_n$ and T be formulae of the first order functional calculus. Then T follows from $A_1,\ldots,A_n$ if and only if the following procedure leads after a finite number of steps to a boolean contradiction:

a. Reduce the set $A_1,\ldots,A_n,\sim T$ of statements to their Herbrand normal form $C_1,\ldots,C_m$.

b. For the variables x_j occurring in $C_1,\ldots,C_m$, substitute formulae belonging to the Herbrand Universe HU constructed from

the constant symbols and function symbols occurring in $C_1,\ldots,C_m$. is leads to a family of substituted clauses C; enumerate the units occurring in these clauses, calling them $P_1,\ldots,P_j,\ldots$, and write the clauses as

$$(6) \qquad C = P_{i_1} \vee P_{i_2} \vee \ldots \vee P_{i_k} \vee \sim P_{j_1} \vee \ldots \vee \sim P_{j_m} .$$

c. As new clauses are generated by the substitution process just described, seek for a Boolean contradiction between these clauses and the collection of all previously generated clauses (6).

The process of substitution and Boolean calculation described by (a), (b) and (c) above is straightforward and obviously algorithmic. Herbrand's theorem tells us that, if we ignore considerations of efficiency and apply this process in an exhaustive manner to an initial set of clauses which is actually inconsistent, a Boolean contradiction must eventually emerge. Of course, if we are to have any practical chance of finding this contradiction, we must use a method more selective than wholesale formula generation. Any theorem-prover algorithm worth considering will in fact incorporate some heuristic for formula selection, which aims to reach a contradiction before the mass of propositions generated becomes overwhelmingly large. Heuristics of at least two types are conceivable. We might imagine heuristics which examine a set of propositions, looking for subsets exhibiting particular patterns, and which, on finding these, route the course of formula generation in particular directions. This corresponds to part of the normal procedure of the mathematician. For example, a mathematician working with formulae describing group theory would probably note that 'transitivity' and 'associativity' both play a role, and, having made this observation, would employ a special notation permitting deduction to proceed in particular directions more rapidly than does the completely general stepwise substitution process that we have just discussed.

Such 'feature noticing' heuristics have in fact not been used much in general theorem-proving algorithms of the kind we shall consider; the development of methods of the kind suggested is probably a useful direction of future work. Algorithmic theorem provers have instead used heuristics of a more general and formal character. Such heuristics can be developed by formal analysis of the manner in which contradictions can be derived. The following reasoning belongs to this line of thought, and will serve to illustrate the meaning of the preceding remark. In deriving a contradiction, we may restrict ourselves to any method of Boolean deduction which is capable of making manifest the contradictory character of an arbitrary set of inconsistent Boolean clauses. It is not hard to see that for this we may rely exclusively on use of the following Boolean rule: from two clauses a $\vee$ B and $\sim$a $\vee$ C, where B and C are themselves subclauses, derive B $\vee$ C. This Boolean rule is sometimes called *resolution*. Now, in the situation with which we mean to deal, the basic terms *a* which can be involved in such resolutions are obtained by substitution of particular elements for the variables occurring in an originally given set of clauses. Given that resolution is the sole Boolean technique that we shall employ, it is clearly not worth making any substitutions which do not lead to the production of a pair of formulae of the form a $\vee$ B and $\sim$a $\vee$ C. On the other hand, given a pair of formulae $a_1 \vee B'$, $\sim a_2 \vee C'$, we can tell algorithmically whether there exists any substitution which, in the necessary sense, will unify a_1 and a_2, i.e., make them identical. For this to be possible, the units a_1 and a_2 must begin with the same predicate symbol, and must be structured in corresponding ways: whenever a_1 and a_2 differ, a variable *x* must be found in one where a variable, constant, or compound structure *e* occurs in the other. The most general substitution which unifies a_1 and a_2 is then the substitution which replaces each such *x* by the corresponding *e*. For example the units

P(x,f(y)) and P(g(z),w)

can be unified by putting g(z) for x and f(y) for w; this is the st general substitution which can be used to make these two units identical.

The above analysis makes it clear that the following set of rules will derive a Boolean contradiction from a set of clauses, if such a contradiction can be derived.

1. Search the clauses for pairs $a_1 \vee B'$, $\sim a_2 \vee C'$ such that a_1 and a_2 can be unified by making some substitution σ for the variables occurring in these clauses.

2. Make this substitution in both clauses, producing two clauses $a \vee B$, $\sim a \vee C$ to which Boolean resolution can be applied.

3. Apply resolution, thus adding a new clause $B \vee C$ to the stock of clauses available.

4. Iterate, until a null clause, i.e., a contradiction, emerges.

The procedure we have outlined, which essentially describes the first formal algorithmic method we shall consider, the so-called *binary resolution method*, will find a contradiction if one exists. In this sense, it is *complete*. Note that the binary resolution procedure bypasses many of the useless false starts which a cruder substitution method might make, only generating substitutions which have some advantage, at least locally. This shows the way in which general formal analysis can suggest improvements in the efficiency of theorem proving algorithms. Of course, the preceding paragraphs have by no means exhausted the measure of formal analysis which may be brought to bear. It is useful, for example, to consider the pattern in which negated and unnegated terms in clauses can enter into the derivation of a contradiction. We will take up this point somewhat below: it will lead us to the so-called *hyper-resolution* variant of the binary resolution method. It may also be observed that, in addition to 'global' heuristics derived by consideration of all the possible ways in which a contradiction might emerge, one may also find 'local' heuristics of a somewhat more *ad-hoc* character to be useful. For example, long clauses will in many cases not be worth considering, since such

clauses must either contain many terms, whose eventual elimination by successive resolutions may be doubtful, or must contain highly substituted and hence very specific terms, which it may be difficult to match with terms in other clauses. For the reasons suggested, a heuristic scheme which counts the number of characters which a clause contains and prefers short to long clauses can be advantageous. We may also wish, as we generate clauses c, to attach to c some measure of the number of steps required to derive c. We might then set aside all c for which this measure grew too large, thereby giving preference to 'shallow' proofs over 'deep' proofs; most of our algorithms are incapable of finding deep proofs in any reasonable length of time, and hence the hint that a proof is shallow is very useful. In point of fact, we shall generally not incorporate heuristics of *ad-hoc* character into the algorithms to be presented in this section. We prefer to omit them in order that the various possible strategies derived from formal analysis which are to be presented can stand forth with maximum clarity.

It should also be noted that the algorithms which follow merely derive a contradiction (if they can) and report their success. Actually, realistic theorem-prover algorithms will keep an account of the parentage of each clause, and, on deriving a contradiction, will print out the ancestry of this contradiction, thus exhibiting the proof that they have found, rather than merely asserting success. As this addition is easily made to the algorithms which are to follow, we have felt free to omit it.

After these general introductory remarks, we begin a more detailed algorithmic discussion. We will always start from a set of clauses which are to be shown to be contradictory. We represent each clause c by a pair

<negunits,posunits> ,

where *posunits* is the set of all units which occur unnegated in c, and *negunits* is the set of all units which occur negated in c.

Each unit is itself taken to be a pair,

<relnsymbol,argsequence> ,

where *relnsymbol* is some (atomic) relation symbol, and *argsequence* is a sequence, namely the sequence of arguments of this relation. Finally, each argument is an element, i.e., is either

i. A variable, represented by an atom;

ii. A constant, represented by a set whose sole member is an atom;

iii. An *item*, represented by a pair
<functionsymbol,argsequence> ,

where *functionsymbol* is some (atomic) function symbol, and *argsequence* is the sequence of arguments of the function, having again one of the forms i, ii, or iii. All this is to say that we represent clauses, and their units and elements, by recursively defined, tree-like structures.

Before coming to the algorithms of principal concern to us, we build up a group of auxiliary routines for the manipulation of logical formulae represented in the form we have just described. First, some elementary algorithms:

```
definef varsof(obj); /* a recursive tree-walker which collects
all the variables of a term or element */
if atom obj then return {obj};;
if type obj ne tupl then return nl;;
/* else */ return [+: arg(n) ε obj(2)] varsof(arg);
end varsof;
```

Next, two subroutines which implement basic substitution operations.

In these basic algorithms (and in certain subsequent algorithms as well) we deal with *substitutions* defined by mappings

(1) map = $\{\langle x_1, e_1\rangle, \langle x_2, e_2\rangle, \ldots, \langle x_n, e_n\rangle\}$.

Each such substitution associates some unique element with each of a finite set of variables x_j. The effect of the substitution (1) on a term is to replace each occurrence of each of the variable x_i by the corresponding element e_i; variables x for which map(x) eq Ω are not replaced. The replacement is 'simultaneous.' Note then that nl acts as the identity substitution.

```
definef obj subst map; /* the basic recursive substitution
     operation, applying either to a term or an element */
return if atom obj then
          if map(obj) is new ne Ω then new else obj
     else if type obj eq tupl then
          <hd obj> + ([+: arg(n) ε obj(2)]<arg subst map> orm nult)
     else obj;
end subst;
```

```
definef mapa compose mapb; /* the 'multiplication' of maps
     corresponding to the preceding substitution operation */
return {<hd x, x(2) subst mapb>, x ε mapa} +
     {item ε mapb | n (hd item) ε hd[mapa]};
end compose;
```

Two units (or elements) *ta* and *tb* are said to be *unifiable* if there exists a substitution *map* such that

(2) (*ta* subst *map*) eq (*tb* subst *map*) .

If such a map exists, then there exists a *most general unifier*,

e., a map m with the property (2) such that any other map with e property (2) can be written in the form m compose n for some n. The map m can be found by the following procedure: *ta* and *tb* must be structurally identical, except that where a variable x occurs in either, some compound element e may occur in the other. If such an x can be found, substitute e for each occurrence of x, thus bringing *ta* and *tb* closer to equality, and continue to search for additional variables x for which substitutions ought to be made. The map we desire is then the composition of all the individual substitutions which this process will uncover. Note however that if x itself occurs as a variable in the element e to be substituted for it, then unification becomes impossible.

The following SETL algorithm makes use of the procedure just described. The algorithm is written as a function which returns the most general unifier of two units (or elements); if unification is impossible, it returns Ω.

```
definef mogenu(ta,tb);
if ta eq tb then return nl;;
if atom ta then return
     if ta ε varsof(tb) then Ω else {<ta,tb>};;
if atom tb then return
     if tb ε varsof(ta) then Ω else {<tb,ta>};;
if n type ta eq tupl or n type tb eq tupl then return Ω;;
if (hd ta) ne hd tb then return Ω;;
argsa = ta(2); argsb=tb(2); map=nl; /*the identity substitution*/
(∀arga(i) ∈ argsa)
     obja = argsa(i) subst map; objb=argsb(i) subst map;
     if mogenu(obja,objb) is mgu eq Ω then
          return Ω;
     else
          map = map compose mgu;
     end if;
end ∀ arga;
return map;
end mogenu;
```

Next we give a routine which forms part of the simplification procedure we will use. A clause c is said to *subsume* a clause c' if there exists a substituted instance $\bar{c}$ of c, such that every positive unit of $\bar{c}$ is included among the positive units of c' and every negative unit of $\bar{c}$ is included among the negative units of c'. In this case it is clear that c represents a more general assertion than c', so that c' may be dropped without loss of information from any set of assertions containing c. Let the units of c' be

$$t'_1 \vee t'_2 \vee \ldots \vee t'_n ,$$

and the units of c be

$$t_1 \vee \ldots \vee t_m .$$

We test for subsumption as follows. Taking t_1, we form the collection of all maps which would make t_1 identical with a term of c'. Then, taking each such map m, we attempt to *extend* it to the next unit t_2 of c, by the following rule. Take t_2, and apply the substitution m to it; call the result E_2. If there is some map m' which makes E_2 identical with a unit of c', the composition m compose m' represents the desired extension of m; if no such m' exists, we simply drop m. Any maps which can be extended to t_2 we attempt to extend to t_3, etc. If there exists any map which can be extended to all the units of c, then c subsumes c'; otherwise not.

Note that in our extension process, we never wish to make a substitution for any variable in a unit of c which prior substitutions have made identical with a unit of c'. This part may be handled neatly by redesignating all the variables of c' as constants before the extension process begins. Given this, the compositions of maps to be performed during the extension process reduce simply to unions of sets of ordered pairs.

Here is the SETL algorithm.

```
definef c subsumes cpr;
cprime = cpr;
varspr = [+: t ε cprime(1) + cprime(2)] varsof(t);
makeconst = {<x,{x}>, x ε varspr};
cprime = <[cprime(1)] subst makeconst,[cprime(2)] subst makeconst>;
maps = {nl}; /* start with set consisting of identity map only */
<neg,pos> = c; <negpr,pospr> = cprime;
(∀tεneg) if extend(maps,t,negpr) is maps eq nl then return f;;;
(∀tεpos) if extend(maps,t,pospr) is maps eq nl then return f;;;
/* if arrive at this point then */ return t;
end subsumes;

definef extend(maps,unit,unitset);
extensions = nl;
(∀m ε maps,t ε unitset)
     if mogenu (unit subst m,t) is newmap ne Ω then
          (newmap + m) in extensions;
     end if;
end ∀m;
return extensions;
end extend;
```

We will also have use for the 'pluralized' variant of the above procedure in which we ask whether any one of a set of clauses subsumes a given clause. This is expressed in SETL simply as follows:

```
definef clauses subsume c;
return ∃ clause ε clauses | (clause subsumes c);
end subsume;
```

Now we come to the resolution process itself. Suppose that, given clauses ca and cb, we can find a negative term na in ca,

and a positive term pb in cb with which na can be unified. Let m be the most general unifier of na and pb. The *resolvent of* ca *with* cb *on* na *and* pb is then obtained by dropping na from ca, pb from cb, joining the remaining terms into a single clause, and applying the substitution m to the result. In SETL, this is:

```
definef resolvent(na,pb,ca,cb);
m = mogenu (na,pb);
hcasub = [ca(1)] subst m; tcasub = [ca(2)] subst m;
hcbsub=[cb(1)] subst m; tcbsub=[cb(2)] subst m;
nasub = na subst m;
return <hcasub less nasub + hcbsub; tcbsub less nasub + tcasub>;
end resolvent;
```

A set s of clauses produced by resolution may contain clauses c subsuming other clauses d in s. It may also contain *tautologous* clauses, i.e., clauses containing a given term both negated and unnegated. Such clauses can clearly never contribute to a contradiction, and may therefore be dropped. The simplification procedure which we now give accepts a set of old and a set of new clauses, removes all subsumed clauses, tautologous clauses, and superfluous units from both sets, and ensures that the variables occurring in any new clauses are disjoint from those appearing in any other clause. This last is a condition assumed implicitly in some of our other algorithms.

```
define simplify(newclauses,oldclauses);
/* olds, news are assumed to be global */
news = newclauses; olds = oldclauses;
(∀clause ∈ newclauses)
     news = news less clause;
     if n superfluous(rebuild(clause) is newclause) then
          newclause  in news;
     end if;
end ∀clause;
```

```
(∀clause ε oldclauses)
     if news subsume clause then
         olds = olds less clause;
     end if;
end ∀clause;
newclauses = news; oldclauses = olds;
return;
end simplify;

definef superfluous(clause);
/* olds, news are assumed to be global */
if olds subsume clause or news subsume clause then return t;;
return ∃ t ε clause(1) | t ε clause(2);
end superfluous;

definef rebuild(clause);
map = nl;
(∀var ε [+: t ε clause(1) + clause(2)] varsof(t)) map(var) = newat;;
return <[clause(1)] subst map, [clause(2)] subst map>;
end rebuild;
```

Having now built up all necessary formula-manipulating auxiliary routines, we go on to describe various of the principal theorem proving methods which appear in the literature. As has been mentioned, theorem-proving methods differ in the strategies which they use to generate new clauses. The first method we shall describe, *binary resolution with set of support*, divides the set of clauses from which it aims to derive a contradiction into two sets: those derived from the axioms of its subject, and those derived by negation of the theorem whose 'proof by contradiction' is to be generated. It then forms pairwise resolvents in all possible ways, never matching two axioms since a contradiction can never be reached through successive resolutions of axioms alone.

Note however that it is essential for the completeness of this method that when forming resolvents for two clauses c and d, we attempt wherever possible to identify two or more units of c (or of d) by applying appropriate substitutions to c. This remark is illustrated by the set

$$a(x) \vee a(c)$$
$$b(x) \vee b(c)$$
$$\sim a(x) \vee \sim b(x)$$

of three clauses. This set is contradictory, but a boolean contradiction can only be derived if the substitution x = c is applied to its first and second clauses. A clause c' produced from the clause c by a substitution which causes two of the units of c to coalesce is called a *factor*, and the term of c' produced by this coalescence is called its *distinguished* term. Of course, when we form a factor we will wish to perform resolution on its distinguished units rather than on any other term, since to proceed otherwise would be to perform work that is bound to be repeated later.

The SETL program for binary resolution theorem proving is as follows (see J. Robinson, "A Machine-Oriented Logic Based on the Resolution Principle," JACM 12, Jan. 1965, pp. 23-41 for the source of this method).

```
definef binres(axioms,antitheorems);
   newer is assumed to be global; cf. matchup */
old = axioms; new = antitheorems;
(while new ne nl)
     newer = nl;
     if t ε (matchup[old,new] /* the procedure matchup will
     generate new resolvents */ + matchup[new,old] +
       matchup[new,new]) then
          return t;
     end if;
     old = old + new; simplify(newer,old);
     new = newer;
end while;
/* if no new resolvents then */ return f;
end binres;


definef matchup(ca,cb);
/* newer is assumed to be global */
(∀x ε tandmaps(ca(1),cb(2))
     <tmx, mapx> = x;
     newres = <[ca(1)]subst mapx less tmx + [cb(1)] subst mapx),
               [cb(2)] subst mapx less tmx + [ca(2)] subst mapx)>;
     if newres = <nl,nl> then
          return t;
     else newres in newer;
end ∀x;
return f;
end matchup;
```

```
definef tandmaps(neg,pos); /* finds all maps which lead to the
     elimination of one or more unit during a resolution; return
     a set of pairs of form <unit eliminated,map> */
     /* first sequence all the units */
nseq = [+: na ε neg]<na> orm nult;
pseq = [+: pb ε pos]<pb> orm nult;
tmps = nl;
(1 ≤ ∀j ≤ #nseq, 1 ≤ k ≤ #pseq | mogenu(nseq(j),pseq(k)) is map ne Ω)
     tm = nseq(j) subst map;
     pile = [{nseq(m),j<m≤#nseq} + {pseq(m), k<m≤ #pseq}] subst map
                                                          less tm;
     work ={<tm,pile,map>};
     (while work ne nl)
          x from work; <tmx,plx,mpx> = x;
          <tmx,mpx> in tmps;
          /* sequence the 'pile' of units */
          seq = [+: t ε plx]<t> orm nult;
          (1 ≤ ∀n ≤ #seq | mogenu(tm,seq(n)) is xtramap ne Ω)
             tmnew = tm subst xtramap;
             pilenew = [{seq(k),n<k≤#seq}] subst xtramap less tmnew;
             mapnew = mpx compose xtramap;
             <tmnew,pilenew,mapnew> in work;
          end ∀n;
     end while;
end ∀j;
return tmps;
end tandmaps;
```

In the theorem proving algorithm just given, the formation factors is inefficient, since factors of a given clause will be formed again and again. We can avoid this inefficiency by restructuring our algorithm somewhat, so as to save factors once formed; this leads to the *factor resolution* method presented below. In this algorithm, we associate, with each clause c, the set *factors(c)* of all its factors. Note also that when a factor of a clause c is produced, we consider the particular one of its units which resulted from the fusion of two units of c to be distinguished, and, when the factor is used to produce a resolvent, insist that this distinguished unit disappear in the resolvent. For this reason factors are saved as triples in the following formats:

<'pos',unit,clause> ,

if they contain a distinguished positive unit, where *unit* is that distinguished unit, and where as usual *clause = <posunits,negunits>* gives the set of all positive and negative units of the factor; similarly, factors containing a distinguished negative unit are represented as triples

<!neg!,unit,clause> .

The SETL algorithm for factor resolution is as follows.

```
definef factres(axioms,antitheorems);
/* factors is assumed to be global */
old = axioms;  new = antitheorems;  factors = nl;
makefacts[old];
(while new ne nl)
   makefacts[new];
   newer = nl;
   if t ∈ (matchfact[old,new] /* the procedure matchfact will
                                 generate new resolvents  */
          + matchfact[new,old] + matchfact[new,new]) then
        return t;
   end if;
   old = old + new;  oldkeep = old;
```

```
    simplify(newer,old);
    (∀x ∈ oldkeep)
        if n x ∈ old then factors(x) = Ω;;
    end ∀x;
    new = newer;
end while;
/* if no new resolvents then */ return f;
end factres;

definef matchfact(ca,cb);
/* newer, factors  are assumed to be global */
/* first match unfactored terms */
(∀na ∈ ca(1), pb ∈ cb(2)  | mogenu(na,pb) ne Ω)
    if resolvent(na,pb,ca,cb) is newres eq <nl,nl> then
        return t;
    else
        newres in newer;
    end if;
end ∀na;
/* now match factored terms */
(∀fct ∈ factors(ca), pb ∈ cb(2) | hd fct eq 'neg')
    <-,dist,caf> = fct;
    if mogenu(dist,pb) is mgu eq Ω then continue;; /* else */
    if resolvent(dist,pb,caf,cb) is newres eq <nl,nl> then
        return t;
    else
        newres in newer;
    end if;
end ∀fct;
(∀fct ∈ factors(cb), na ∈ hd ca | hd fct eq 'pos')
    <-,dist,cbf> = fct;
    if mogenu(dist,pb) is mgu eq Ω then continue;; /* else */
    if resolvent(na,dist,ca,cbf) is newres eq <nl,nl> then
        return t;
    else
        newres in newer;
    end if;
end ∀fct;
```

```
/* if arrive at this point then */ return f;
  d matchfact;

define  makefacts(cℓ);  /*builds up all factors of a clause*/
/* factors is assumed to be global */
factors(cℓ) = {<'neg'>+fct, fct ∈ negfacts(cℓ)}
            + {<'pos', fct(1),<fct(2)(2),fct(2)(1)>,
                 fct ∈ negfacts(cℓ(2),cℓ(1)>)};
return;
end makefacts;

definef negfacts(cℓ);
/* builds factors by identifying 'negative' units */
<negu,posu> = cℓ;  facts = nℓ;
/* make sequence of 'negative' units */
seq = [+: x ∈ negu]<x> orm nult; work = nℓ;
(1 ≤ ∀j ≤ #seq, j < k ≤ #seq|mogenu(seq(j),seq(k)) is map ne Ω)
   ut = seq(j) subst map;
   pile = [{seq(ℓ), k < ℓ ≤ #seq}] subst map;
   clprime = <[cℓ(1)] subst map, [cℓ(2)] subst map>;
   <ut, pile, clprime> in work;
end ∀j;
(while work ne nℓ)
   x from work; <ut,pile,clprime> = x;
   <ut,clprime>   in facts; /* sequence the 'pile' of units */
   seq = [+: x ∈ pile]<x> orm nult;
   (1 ≤ ∀j ≤ #seq|mogenu(ut,seq(j)) is map ne Ω)
        ut = ut subst map;
        pil = [{seq(k), j<k<#seq}] subst map;
        clprim = <[clprime(1)] subst map,[clprime(2)]subst map>;
        <ut,pil,clprim> in work;
   end ∀j;
end while;
return facts;
end negfacts;
```

Clauses may be distinguished as *positive* (containing positive clauses only) and *mixed* (containing both negative and positive clauses, or negative clauses only). The clause types which can result on forming the resolvent of a pair of clauses are as follows:

positive with mixed: null, positive, or mixed
mixed with mixed: mixed .

Note in particular that a null clause, i.e., a visible contradiction, can never result from two mixed clauses, but only from a positive and a negative clause. This remark suggests that positive clauses deserve special attention. Next observe that the process of resolution has a kind of 'associativity' property. If the resolvent c of two mixed clauses ca and cb is produced, and the resolvent d of c with a positive clause p is produced, then the term entering into the second resolution must be derived from a term either of ca or of cb, say ca for the sake of definiteness. We could therefore obtain the same resolvent by resolving ca with p, and then resolving cb with the result. Now, if d is positive (or null) it follows that the resolvent of ca with p must be positive. Arguing inductively, we see that every positive (or null) clause can be produced by a chain of resolutions such that one of the two clauses entering into each resolution is positive. Thus, in searching for a contradiction, we may restrict ourselves to forming resolvents from pairs of clauses, one of which is positive.

This last remark leads to yet another observation. One principal difficulty which any resolution theorem prover must attempt to overcome is the tendency for clauses to accumulate rapidly during the functioning of the algorithm, leading eventually to swamping of available memory. Given that we can proceed by resolving positive with mixed clauses exclusively, it is clear that, if we are willing to resolve a mixed clause against an entire sequence of positive clauses, we need not bother to accumulate new mixed clauses, but can accumulate positive clauses only. If no new positive clauses can be

accumulated, it follows that the set of clauses with which our algorithm is working is consistent. Resolution theorem proving by the use of this strategy was introduced by J. Robinson, cf. Automatic deduction with hyper-resolution, Int. J. Comp. Math. 1 (1965) pp. 227-234, and is called *hyperresolution*. Note also that if a new positive clause can result by successive resolutions of a sequence of positive clauses with a given mixed clause m, it follows that all the negative terms of m must eventually be eliminated. Thus we may attempt to match the negative terms of m in any convenient order.

The following SETL algorithm embodies the theorem-proving strategy just outlined; note that the preceding routine *negfacts* is used, and that this returns a set of pairs

<nt,<negcl,poscl>>

in which *nt* is a distinguished negative unit, while *negcl* and *poscl* are respectively the negative and positive units of a clause containing this unit and obtained by factorization.

```
   define hyperres(axioms,antitheorems);
   all = axioms + antitheorems;
   setpos = {c ε all | c(1) eq nl};
   setmixed = all - setpos;
try:(∀mc ∈ setmixed) /* try to form new positive clauses using mc */
       work = {mc};
       (while work ne nl doing work = rebuild[nework];)
            nework = nl;
           (∀mcl ε work)
              /* first form negative factors and match against
                       positive clauses */
              nfacts = negfacts(mcl);
              (∀nf ε nfacts,poscl ε setpos,post ε poscl(2) |
                        mogenu(nf(1),post) ne Ω)
                 <dist,cl> = nf;
                 if resolvent(dist,post,cl,poscl) is newres eq <nl,nl>
                       then print 'proof of contradiction obtained';
                       return;
                 else if (hd newres) eq nl then
                       newres in newpos;
                 else
                       newres in nework;
                 end if;
              end ∀nf;
              /* next treat factors of positive clauses */
              negt = mcl(1);
              (∀poscl ε setpos)
                 pfacts = negfacts<poscl(2),nl>;
                 (∀pf ε pfacts,nt ε negt | mogenu(nt,pf(1)) ne Ω)
                       <dist,ptms> = pf;
                        pcl = <nl, hd ptms>;
                       if resolvent(nt,dist,mcl,pcl) is newres
                          eq <nl,nl> then
                             print 'proof of contradiction obtained';
                       else if (hd newres) eq nl then
```

```
                    newres in newpos;
                else
                    newres in nework;
                end if;
            end ∀pf;
        end ∀poscℓ;
        /* now process unfactored clauses */
        (∀poscℓ ε setpos,post ε poscℓ(2),nt ε negt |
                    mogenu(post,nt) ne Ω)
            if resolvent(nt,post,mcℓ,poscℓ) is newres
                    eq <nℓ,nℓ> then
                print 'proof of contradiction obtained';
                return;
            else if hd newres eq nℓ then
                    newres in newpos;
            else
                    newres in nework;
            end if;
        end ∀poscℓ;
    end ∀mcℓ;
end while;
simplify(newpos,setpos);
if newpos ne nℓ then
    setpos = setpos + newpos;
    go to try;
end if;
end ∀mc;
/* if this point is reached, no new positive clauses can be formed */
print 'given set of clauses is consistent';
return;
end hyperres;
```

Resolution theorem provers will often exhibit a kind of indecisiveness, forming a promising clause from a given clause c but then not pursuing the line of investigation opened up. Our next method, *maximal clash resolution*, seeks to overcome this difficulty by performing resolution repeatedly on the terms of a given clause as often as possible before a new clause is taken up. (Cf. J. Robinson, A review of automatic theorem proving, Proc. Symposia in Appl. Math. (1966), v. 19, American Mathematical Society, Providence, R. I. for the source of this approach.) In the SETL algorithms which follow, the following convention is used to facilitate the processes that must be applied. As resolution is repeatedly applied to the terms derived from an original clause cℓ, the clauses which result are maintained as triples in the form

<<ntmscl,ptmscl>,extneg,extpos> ,

Here *ntmscl* are all the negative terms, and *ptmscl* all the positive terms, which derive from terms originally present in *cl;* while *extneg* are all those other negative terms, and *extpos* all those other positive terms, which derive from clauses against which *cl* has been matched for the forming of resolvents.

The SETL algorithm for maximal clash resolution is as follows:

```
definef maxclash(axioms,antitheorems);
/* old, proved, and newer are assumed to be global */
new = axioms + antitheorems;  old = new;  proved = f;
(while new ne nl)
   newer = maxresols[new];
   if proved /*proved is a flag set by maxresols*/ then
        return t;
   end if;
   old = old+new;
   simplify(newer,old); new=newer;
end while;
/* if no new resolvents then */ return f;
end maxclash;
```

```
definef maxresols(clause);
   old,proved,newer,work,canextend, and map are assumed to be global*/
if proved then return nl;;  /* bypass work if finished already */
work = {<clause, nl, nl>};  maxs = nl;
(while work ne nl)
   elt from work;
   <<ntmscl,ptmscl>, xtrneg, xtrpos> = elt;
   canextend = f;
   vars = [+: x ∈ ntmscl + ptmscl + xtrneg + xtrpos]varsof(x);
   map = nl;
   (∀v ∈ vars) map(v) = newat;;
   ntmscl = substin ntmscl;  ptmscl = substin ptmscl;
   xtrneg = substin xtrneg;  xtrpos = substin xtrpos;
   extnegcl(ntmscl,ptmscl,xtrneg,xtrpos,'neg');
   extnegcl(ptmscl,ntmscl,xtrpos,xtrneg,'pos');
   if canextend then continue;; /* else */
   <ntmscl + xtrneg, ptmscl + xtrpos> is newresol in maxs;
   if newresol eq <nl,nl> then
        proved = t;  return nl;
   end if;
end while;
return maxs;
end maxresols;

/* auxiliary substitution routine */
definef substin x;
/* map is assumed to be global */
return[x] subst map;
end substin;

define  extnegcl(ntmscl,ptmscl,extneg,extpos,switch);
/* extends clash with negative terms of clause */
/* map, work, canextend, and old are assumed to be global */
(∀oldcl ∈ old)
   if switch eq 'neg' then <ont,opt> = oldcl;
                      else <opt,ont> = oldcl;;
   (∀x ∈ tandmaps(ntmscl,opt))
```

```
/* tandmaps will produce a set of pairs <term,map>,
each showing a map which may be used to produce a
resolvent and the term which resolution will then
eliminate; the code for tandmaps has been given above */
     <term,map> = x;
     ntmsc = substin ntmscl;
     ptmsc = substin ptmscl;
     extng = substin extneg;
     extp = substin extpos;
     ontl  = substin ont;
     optl =  substin opt;
     ntmsc = ntmsc less term;
     extng = extng + (ontl - ntmsc) less term;
     extp =  optl + (extp - ptmsc) less term;
     if n ∃at ∈ ntmsc + extng | at ∈ (ptmsc + extp) then

        canextend = t;
        if switch eq 'neg' then
           <<ntmsc,ptmsc>,extng,extp> in work;
        else
           <<ptmsc,ntmsc>,extp,extng> in work;
        end if switch;
     end if n ∃;
   end ∀x;
end ∀oldcl;
return;
end extnegcl;
```

The partial "associativity" of the resolvent-forming process, which we have already alluded in our discussed of hyper-resolution, can be used in additional ways to standardize any tree T of resolutions which leads to a contradiction, thereby cutting down on the number of alternatives which a theorem-prover algorithm must explore. In particular, suppose that a contradictory set s of clauses is given, but that there exists a subset n such that s-n is consistent. Then, by using the above-mentioned partial "associativity" to force T into a pattern in which clauses are resolved against clauses of s (input clauses) as often as possible, it may be shown that we can always assume T to have a special form, which we may describe as follows (Cf. R. Anderson and W. Bledsoe, A Linear Format for Resolution with Merging and a New Technique for Establishing Completeness, J. ACM 17 (3), 1970, pp. 525-534, for a proof).

Given two clauses c_1 and c_2 containing unifiable terms t_1, $\sim t_2$, and given the map m which unifies these terms and the resolvent c produced from this unification, we call a term t of c a *merge term* if it arises both as $\bar{t}_1$ subst map and $\bar{t}_2$ subst map, where $\bar{t}_1$ is a term of c_1 and $\bar{t}_2$ is a term of c_2. Then, in the situation described above, T may be taken to consist of a sequence of resolutions, the i+1'st resolution producing a clause R_{i+1} from two clauses R_i and C_i, and where the following assertions hold.

i. Each C_i is either in s (i.e., is one of an original 'input' set of propositions) or is an R_j with $j < i$;

ii. If C_i is not in s, then C_i contains a merge term t, and the term resolved upon to produce R_{i+1} from R_i and C_i is t. Moreover, every term of R_{i+1} is a term of an appropriately substituted version of R_i (subsumption condition).

We note finally that the initial clause R_0 may be an arbitrarily specified element of the initially given set of clauses.

The assertions made in the preceding paragraphs establish the completeness of the modified resolution procedure, which may be called the *linear resolution* method, embodied in the SETL algorithm below. Note that during the i-th iteration of this algorithm (we refer here to the outermost 'while' loop) the set *curres* represents all the resolvents R which might appear in the i-th place in a sequence of resolvents $R_0,\ldots,R_i$ satisfying the above conditions and with R_0 = c; except that if R would appear with lower index in such a sequence we drop it from *curres*. The set *allres* represents all the resolvents R which could ever appear in such a sequence, only clauses which are properly subsumed by clauses subsequently appearing in *curres* being dropped. The set *mergeres* represents the cumulative total of all resolvents belonging to such a sequence and containing a merge term.

The routine *matchup2*, modeled after the *matchup* routine used in the binary resolution algorithm given earlier, not only forms additional resolvents but notes those resolvents which are merge resolvents; merge resolvents are represented as pairs <<negmerge,posmerge>, cc>, where *cc* is a clause containing merge terms, *negmerge* is the set of the negative merge terms which c contains, and *posmerge* is the set of positive merge terms which c contains. The routine ca subsinst cb tests to see whether there exists a *map* such that every term of ca is included among the terms of *cb* subst *map*. The technique is rather like that used in the subsumes routine given earlier: All the variables of ca are redesignated as constants; a set of *maps* is initialized to consist of the identity map only. Then, proceeding through the terms t of ca successively, and for each *map* $\in$ *maps* , one tests to see if *map* can be extended to a *map'* in such a way as to make t agree

with a term of *cb* subst *map'*. If a sequence of extensions
timately covering every term t of ca can be made, true is
returned; otherwise false .

Here is the SETL algorithm.

```
definef linres(axioms,antitheorems);
/* mergeres, newer, cl, and clm are assumed
   to be global */
input = axioms + antitheorems;
c = ∋input;  /* choose some starting clause */
curres = {c};   allres = curres;
mergeres  = nl;
(while curres ne nl)
/* as long as there are current resolvents */

   (∀cl ∈ curres)
        <neg,pos> = cl;
        (∀item ∈ mergeres)
             <<negmrg,posmrg>,clm> = item;
   if matchup2(neg,posmrg,cl,clm) or matchup2(negmrg,pos,clm,cl)then
                  return t;
             end if;
        end ∀item;
   end ∀cl;
   clm = Ω;
/* to control a detail of the processing by matchup2;
   note that matchup2 will define a new total set of resolvents,
   and a new set of merge resolvents */
   (∀cl ∈ allres)
        <neg,pos>. = cl;
        (∀clin ∈ input)
             <negin,posin> = clin;
if matchup2(neg,posin,cl,clin) or matchup2(negin,pos,clin,cl)then
                  return t;
             end if;
        end ∀clin;
   end ∀cl;
```

```
    simplify(newer,allres);
    allres = allres + newer;
    curres = newer;
end while;
/* if no surviving resolvents then */
return f;
end linres;

definef matchup2(neg,pos,ca,cb);
/* forms resolvents and notes merge resolvents */
/* note use of subroutine tandmaps given previously */
/* mergeres, newer, cl,  and clm are assumed to be global */
(∀x ∈ tandmaps(neg,pos))
    <tmx,mpx> = x;
    newres = <ca(1) subst mapx less tmx is n1
                + (cb(1) subst mapx is n2),
          cb(2) subst mapx less tmx is p1 + (ca(2) subst mapx is p2>;

    if newres eq <nℓ, nℓ> then return t;;
    if n clm eq Ω /* so that not matching an input clause
                     but an old merge resolvent */  then
         if n(newres subsinst clm) then continue;;
    end if;  /* else */
    newres in newer;
    mergeparts = <n1 * n2, p1 * p2>;
    if mergeparts ne <nℓ,nℓ> then

         mergeres = mergereswith <mergeparts,newres>;

    end if;
end ∀x;
return f;
end matchup2;
```

```
definef ca subsinst cb;
   determines whether there exists a map such that every term of ca
   is included among the terms of cb subst map */
varsa = [+: t ∈ ca(1) + ca(2)] varsof(t);
makconst = {<x,{x}>, x ∈ varsa};
<nega,posa> = <[ca(1)] subst makconst,[ca(2)] subst makconst>;
maps = {nl}; /* start with set consisting of identity map only */
<negb,posb> = cb;
(∀ta ∈ nega)
   if extend2(maps,ta,negb) is maps eq nl then return f;;
end ∀ta;
(∀tb ∈ posa)
   if extend2(maps,tb,posb) is maps eq nl then return f;;
end ∀tb;
/* if arrive at this point then */ return t;
end substinst;



definef extend2(maps,term,termset);
/* auxiliary routine which tests to see if by extending some
   map ∈ maps to a newmap, term can be identified with an element
   element of termset subst map subst newmap */
extensions = nl;
(∀m ∈ maps, t ∈ termset)
   if mogenu(t subst m, term) is addition ne Ω then
        addition + m in extensions;
   end if;
end ∀m;
return extensions;
end extend2;
```

D. Loveland has given a theorem-proving algorithm rather closely related to the linear resolution method discussed just above; Loveland calls his procedure *model elimination*. (See Loveland, A simplified format for the model elimination theorem-proving procedure, J. ACM v. 16, pp. 349-363 (1969).) Like the linear resolution method just described, model elimination forms resolvents using input clauses whenever possible. The 'merge resolvent' case arising in the preceding algorithm is handled somewhat differently, however. The technique used is as follows. As resolvents are formed, a partial record of their parentage is kept. Resolvents are in fact maintained as sequences $t_1,\ldots,t_n$ of units (Loveland calls these sequences 'chains'). Certain of the units in a sequence are specially flagged. A sequence corresponds, in the more customary resolution theorem provers, to a clause formed by a succession of resolutions; flagged units are those which resolution would have eliminated. In Loveland's procedure, these units are kept (but specially marked) so as to permit the subsequent elimination of additional units by a logical equivalent of the 'factoring' process applied in conventional resolution theorem provers. It is not hard to see that if resolvents are kept in this form the following three processes must be applied:

1. Extension (Corresponding to simple resolution). Take a sequence $t_1,\ldots,t_n$ whose last unit is unflagged. Find an input clause c and a unit t in c, having negativity opposite to that of t_n, and such that t and t_n can be unified by a map m. Apply the map m to all the terms $t_1,\ldots,t_n$ and to all the terms of c, thus obtaining a new sequence $t_1',\ldots,t_n'$ and a new clause c'. Eliminate t map m from c' and append the remaining units of c to $t_1',\ldots,t_n'$; flag t_n'.

2. Reduction (Corresponding to factoring). If a sequence $t_1,\ldots,t_n$ contains a flagged unit t_j which is followed by an unflagged unit t_k of opposite negativity, and if there exists a map m which unifies t_j and t_k, then apply m to all the units of $t_1,\ldots,t_n$, and drop the k-th unit of the result.

3. Contraction (Dropping terms no longer needed). If a sequence ends with a flagged unit, drop this unit.

Process 3 should be applied when possible; if this is not possible, then process 2; otherwise process 1. A contradiction is represented by a sequence of length zero. A sequence containing a flagged unit identical (and in particular having the same negativity) to a unit, flagged or unflagged, standing to its right represents a resolvent into whose formation a tautologous clause has entered; such a sequence may therefore be ignored. A sequence containing two flagged identical units of opposite negativity represents a resolvent into whose formation a clause with duplicate literals has entered and may therefore be ignored also. Finally, a sequence containing two unflagged identical terms of opposite negativity between flagged units comes from a tautology, and may be ignored for this reason.

In the following SETL algorithm, the 'sequences' spoken of above will be represented as sequences of triples

<unit,posneg,flag>

where *posneg* is t if the unit is to be considered as unnegated, f otherwise; *flag* is t if the unit is to be considered as flagged, f otherwise.

In our algorithm, the following macro will be used when a mapping *map* which unifies two terms of opposite negativity has been discovered, and a new sequence of terms must be built by appending sets *mapneg* of new negative and *mappos* of new positive terms to an old sequence *s*:

```
macro buildnewseq;
newseq = [+: s(y) ∈ seq]<<hd s subst map> + tl s>
newseq(#seq)(3) = t /* 'flag' final unit of sequence */
(∀ ut ∈ mapneg) newseq(#newseq+1) = <ut,f,f>;;
(∀ut  ∈ mappos) newseq(#newseq+1) = <ut,t,f>;;
if trim(newseq) then
     return t;
else if n ignore then
     newseq in newchains;
end if;
endm buildnewseq;
```

```
definef modlim(axioms,antitheorems);
/* ignore is assumed to be global */
all = axioms + antitheorems;  chains = nl;
(∀x ∈ antitheorems) <neg,pos> = rebuild(x);
   seq = [+: nu ∈ neg]<<nu,f,f>> orm nult;
   (∀pu ∈ pos) seq(#seq+1) = <pu,t,f>;;
   seq in chains;
end ∀x;
(while chains ne nl)
/* first apply reduction process to produce additional chains*/
   redchains = chains;
   (while redchains ne nl)
        newred = nl;
        (∀seq ∈ redchains, s(j) ∈ seq, j < k ≤ #seq |
             s(3) and n seq(k)(3) and s(2) ne seq(k)(2)
             and mogenu(hd s, hd seq(k) is map ne Ω)
             newseq=[+: 1≤l≤j]<<hd seq(l)subst map>+ tl seq(l)>;
             utm = hd s  subst map;
             (j < ∀l ≤ #seq)
                  t = seq(l);
                  utlm = hd t subst map;
                  if utlm eq utm and n t(3) and s(2) ne t(2) then
                       continue;
                  end if;
                  /* else */ newseq(#newseq+1) = <utlm>+ tl t;
             end ∀l;
             if trim(newseq) then return t;;  /* else */
             if n ignore  /* ignore is global and set by trim*/then
                  newseq in chains;
                  newseq in newred;
        end ∀seq;
        end if;
        redchains = newred;
   end while redchains;
   /* now apply extension process */  newchains = nl;
   (∀seq ∈ chains, incl ∈ all)
        if seq(#seq)(2) then go to postlast;;
```

```
        /* else negative unit is last */
        (∀pu ∈ incl(2)|mogenu(pu, hd seq(#seq)) is map ne Ω)
             <mapneg,mappos> =
             <[incl(1)] subst map, [incl(2)] subst map
                  less (pu subst map)>;
             buildnewseq;
        end ∀pu; /* completes treatment of trailing negative unit;
             treatment of trailing positive unit follows */

[poslast:] (∀nu ∈ hd incl|mogenu(nu,hd seq(#seq))) is map ne Ω)
             <mapneg, mapos > =
             <[incl(1)] subst map less (nu subst map),
                  [incl(2)] subst map>;
             buildnewseq;
        end ∀nu;
   end ∀seq;
   chains = newchains;
end while chains;
/*  if reach this point then */  return f;
   end modlim;

definef trim(seq);
/* ignore is assumed to be global */
(while seq(#seq)(3))
   seq(#seq) = Ω;
   if seq eq nult then return t;;
end while;
/* now check for various conditions of redundancy */
ignore = f;
if 1 ≤ ∃j < #seq, j < k ≤ #seq |
        seq(j)(3) and seq(j)(1:2) eq seq(k)(1:2)
        or  (seq(j)(3) and   seq(k)(3) and hd seq(j) eq hd seq(k)
        and seq(j)(2) ne seq(k)(2)) then
   ignore = t;
   return f;
    if;
```

```
if 1 ≤ ∃j ≤ #seq, j<k≤#seq, k<ℓ≤#seq, ℓ≤m≤#seq |
   seq(j)(3) and seq(m)(3) and seq(k)(1) eq seq(ℓ)(1)
     and seq(k)(2) ne seq(ℓ)(2) and n seq(k)(3) and n seq(ℓ)(3)then
   ignore = t;
   return f;
end if;
/* if not redundant, then substitute new variables before return*/
vars = [+: s(j) ∈ seq] varsof(hd s); map = nℓ;
(∀v ∈ vars) map(v) = newat;;
seq = [+: s(j) ∈ seq] <<(hd s) subst map> + tℓ s>;
return f;
end trim;
```

9 Some artificial intelligence algorithms.

Two-person competitive board games are often objects of study in investigations of artificial intelligence. The small, self-enclosed world of such a game is a maximally simple setting in which to analyze the action of algorithms which detect logical patterns and choose effectively among alternatives. Such games fall into two classes, the *deterministic*, in which the board position evolves strictly in response to alternating moves of the two players; and the *chance games*, in which the board position is also affected by some randomly varying quantities (for example, by dice or cards). We consider deterministic games. Any such game will at any time be in a given board *position*. Each position will define a set of possible *moves*; and a given move, if chosen, will bring the board to some well-defined *new position*. The two players will choose moves alternately, always from a set *possmoves(posn)* determined by the current board position *posn*. The rules of the game will define a set of *terminal positions*, and associate a *payoff* (or *prize*) value with each such position.

We define the value of a given nonterminal position *posn*, to the player P having the move in that position, to be the maximum payoff which he can secure, in spite of his opponent's best efforts, by choosing his future moves correctly starting from *posn*. This is of course equal to the amount which his opponent Q is certain to lose, if play starts in the position *posn*, P having the move and being assumed to play 'perfectly'. Call this quantity *value(posn)*, and let *newposn(posn,m)* be the new position which will result if the (legal) move m is executed in position *posn*. If P chooses move m, then he will arrive at a position from which Q's payoff (assuming perfect play on both sides) is

(1) value(newposn(posn,m)) ,

and hence a position in which his own payoff (on the same assumption) will be

(2) -value(newposn(posn,m)) .

best new position available to P (after the move which he must make) is of course that in which (2) is maximized; and the maximum of (2) clearly is the value to P of the position *posn* from which

he must move. Thus we have

$$(3)\quad \text{value}(\text{posn}) = \max_{m \in \text{possmoves}(\text{posn})} (-\text{value}(\text{newposn}(\text{posn},m))) \ .$$

The position-value function *value* is determined by equation (3) and by the condition that in each terminal position value(posn) is the payoff (by the player who has just moved, to his opponent) determined by the rules of the game being considered.

It is plain from the preceding discussion that, when it is his turn to move, player P should choose a move m which maximizes the expression (2); any such move will be as advantageous to him as any other. This rather simple rule is a universal recipe for perfect play in any board game whatsoever. It is however generally impractical to apply, since, for most interesting games, no rapid scheme for the calculation of the value function is known, and because complete tables of this function would be very large. In checkers, for example, the number of logically possible positions is approximately 10^{19}. The way around this difficulty often adopted in game-playing programs, is as follows. One chooses some easily calculated function *approxvalue(posn)* which is known (or suspected) from experience with a game under investigation to be a monotone increasing function of the optimal strategy-determining function *value*. In the game of chess, for example, this would be an artfully devised combination of scores assigned to such factors as pieces remaining on the board, mobility of pieces, center control, pawn configuration, and so forth. Putting

$$(4)\quad \text{value}_0(\text{posn}) = \text{approxvalue}(\text{posn}) \ ,$$

and using the recursive definition

$$(5)\quad \text{value}_d(\text{posn}) = \max_{m \in \text{possmoves}(\text{posn})} (-\text{value}_{d-1}(\text{newposn}(\text{posn},m)))$$

one then develops a better (cf. (3)) approximation to the true *value* function. Finally, one chooses that move m for which

$$(6)\quad -\text{value}_{d-1}(\text{newposn}(\text{posn},m))$$

is a maximum.

All this is shown in the following straightforward SETL gorithm, in which the function *valued* calculates the most advantageous move at the same time that it accomplishes the calculation (5). The parameter *maxdepth* is the *d* of (5); the trivial function *bestmove*, given first, returns the recommended move as its result. Note the use of the function orm defined on p. 216.

```
scope best;  global depth, move, maxdepth, approxvalue,
   possmoves, newposn;
owns bestmove(depth);

definef bestmove(posn,maxdepth);
depth = 0;
posvalue = valued(posn); /* this function call establishes
   the value of move */
return move;
end bestmove;
end best;

definef valued(posn);
include best*;
if depth eq maxdepth then return approxvalue(posn);;
bestilnow = Ω;
(∀m ∈ possmoves(posn))
    depth = depth + 1;
    valmove = valued(newposn(posn,m));
    depth = depth-1;
    if valmove gt (bestilnow orm (valmove-1))
    /* since Ω represents 'minus infinity' */
        then bestilnow = valmove;
            if depth eq 1 then /* save best move */ move = m;;
    end if valmove;
end ∀m;
return bestilnow;
end valued;
```

The algorithm shown above can be improved in a significant way (by using the so-called '(α,β)-cutoff'). If a position p_1 of final estimated value at least v (to me) is reached from an initial position p_0 via a position p_2 in which my oponent has the move and which is of final estimated value at least u (to my opponent) and if $-v \leq u$, then we can be sure that p_1 is not relevant to the choice of move in position p_0. Indeed, since at the intermediate position p_2 my opponent has a line of play yielding him an estimated outcome u, he will not allow himself to be pushed from p_2 into a position p_1 of estimated value to him certainly not exceeding -v. For the same reason, the potential line of play $p_0 \cdots p_2 \cdots p_1$ may be ignored in calculating $value_d(p_0)$. It follows that examination of positions developing from p_1 can be broken off as soon as an estimated value in excess of -u is ascribed to p_1.

The modifications in our earlier algorithms necessary to incorporate this important refinement are shown below. Note that *valued* becomes a function of three parameters. The first of these parameters is a board position. The second parameter is the maximum value which the player having the move can hope for (given other options open to his opponent on the path leading to a given position); this is essentially the quantity -u of the preceding remarks. The third parameter, carried so that it can be passed down a level, is the value which the player having the move is sure of attaining (in view of already-explored options open to him along the path to a given position). Initially, the revised procedure *valued* should be called with second parameter equal to $+\infty$ and third parameter equal to $-\infty$; in the SETL code which now follows, these extreme values are instead represented by Ω.

```
scope best;  global depth,move,maxdepth,
approxvalue, possmoves, newposn; owns bestmove(depth);

definef bestmove(posn,maxdepth); /* revised bestmove function */
depth = 0;
posvalue = valued(posn,Ω,Ω); /* this function call establishes
   the value of move */
return move;
end bestmove;
end best;
```

```
definef valued(posn,mymax,mymin);
   lude best*;
if depth eq maxdepth then return approxvalue(posn);;
bestilnow = mymin;
(∀m ∈ possmoves(posn))
    depth = depth+1;
    valmove = -valued(newposn(posn,m), -bestilnow, -mymax);
    depth = depth-1;
    if valmove gt (mymax orm (valmove+1))
    /* since here Ω represents 'plus infinity' */
        then return valmove;;
    if valmove gt (bestilnow orm (valmove-1))
    /* since here Ω represents 'minus infinity' */
        then bestilnow = valmove;
            if depth eq 1 then /* save best move */ move = m;;
    end if valmove;
end ∀m;
return bestilnow;
end valued;
```

The relative advantage of the modified over the unmodified algorithm can be seen most readily by considering the ideal case in which *approxvalue(posn)* is identical with the true game value function defined by (3), and in which the iteration over *possmoves(posn)* in the algorithms shown above always examines the move m for which valued(newposn(posn,m)) takes on its maximum value before any other node is examined. It can be shown in this case that, if called with depth parameter d to a tree in which T moves are possible in every position, the first algorithm will examine approximately T^d nodes, whereas the second algorithm will examine approximately $T^{d/2}$ nodes. Thus, in this most advantageous case, the improved algorithm will allow one to look ahead twice as many levels with given computational effort as will the unimproved algorithm.

This suggests a procedure in which we

make a preliminary 'shallow' estimate of the relative advantage of various moves, and then sort the moves into order of decreasing estimated advantage before proceeding with the systematic investigation of their consequences. We are thus led to the following further modification of the move-choice and of the recursive position-evaluating functions. In the algorithms shown below sort designates a procedure used to sort moves into order of decreasing estimated advantage (any one of the fast sort procedures described in Item 14, Section 5B may be used), and in which the parameter *surveydepth* determines the depth of the preliminary survey made to estimate move desirability. Note that in the algorithms which follow this preliminary survey is omitted whenever one deals with tree nodes lying within distance *surveydepth* of the stated maximum depth of investigation of moves.

```
scope best; global depth, move, maxdepth, surveydepth,
   approxvalue, possmoves, newposn; owns bestmove(depth);
definef bestmove(posn,maxdepth,surveydepth); /* second revision*/
depth = 0;
posvalue = valued(posn,Ω,Ω); /* this function call establishes
   the value of move */
return move;
end bestmove;
end best;

definef valued(posn,mymax,mymin);
include best*;
if depth eq maxdepth then return approxvalue(posn);;
if depth gt (maxdepth-surveydepth) or surveydepth eq 0 then
   /* don't make preliminary survey; establish arbitrary
   order of moves */
   ordermoves = [+: m ∈ possmoves(posn)]<m>;
else
```

```
    /* reset depth parameter to force survey to specified depth */
    keepdepth = depth; depth=maxdepth-surveydepth+1;
    surveytupl=[+: m∈possmoves(posn)]<<m,valued(newposn(posn,m),Ω,Ω)>>;
    depth = keepdepth; /* restore true depth value */
    /* now sort surveytupl  in order of decreasing values
       of first components of entries */
    sort surveytupl;  /* the routine sort will not be shown */
    ordermoves = [+: c(n) ∈ surveytupl]<hd c>;
end if depth;
/* now make finer estimate of position value */
bestilnow = mymin;
(∀m(n) ∈ ordermoves)
    depth = depth+1;
    valmove = -valued(newposn(posn,m),-bestilnow,-mymax);
    depth = depth-1;
    if valmove gt (mymax orm (valmove+1))
    /* since here Ω represents 'plus infinity' */
        then return valmove;;
    if valmove gt (bestilnow orm (valmove-1))
    /* since here Ω represents 'minus infinity' */
        then bestilnow = valmove;
        if depth eq 1 then /* save best move */ move = m;;
    end if valmove;
end ∀m;
return bestilnow;
end valued;
```

If the estimation function *approxvalue(posn)* supplied to the preceding algorithm is a relatively perfect approximation to the true position-value function associated with a game, and if T moves are possible in each position, then the algorithm shown just above will be able to look ahead d moves in a time roughly proportional to $T^{(d+sd)/2}$, sd denoting the *surveydepth* value that is used. The better the approximation function, the more reasonable it is to use a *surveydepth* value close to 1. However, such good approximations are rarely available for the most interesting games.

A well thought out position value estimation function will measure the (largely 'strategic') value of stable positions reasonably well, but will not respond with equal sensitivity to the tactical aspects of highly 'dynamic' positions in which strong 'threats' play important roles. Thus it can be of advantage to classify positions into 'static' and 'dynamic' categories, and to be more careful in calculating the value of a dynamic than of a static position. We may consider a position to be dynamic if it allows the player having the move to increase the estimated value of his position substantially, or if only a very few moves which don't diminish the estimated value of his position substantially are available to this player.

As a final algorithm in our series of move-choice procedures, we give a version of the *bestmove/valued* pair which looks more deeply into dynamic than into static positions. The following remarks will aid in the comprehension of these revised algorithms.

i. Since *valued*, as revised, manipulates the quantities *depth* and *maxdepth* in a manner which is inherently complex, these quantities appear as explicit parameters of the recursive *valued* procedure; in the preceding versions of this procedure, these quantities were manipulated only in very simple ways, allowing them to be (unstacked) global variables instead.

ii. In the algorithm, shown below, as in the immediately preceding algorithm, a preliminary survey of positions is made to estimate move desirability, allowing deeper examination of moves to be made in order of decreasing desirability. The parameter *surveydepth* shown below is a mapping, giving the depth of this preliminary survey as a function of the depth of tree search already attained. Naturally, the deeper we have already come in examining developments from a given position, the less additional search depth we will be able or willing to tolerate.

iii. Similarly, the parameter *biggermax* is a mapping giving the revised depth to which our algorithm will proceed when it encounters active positions at what would normally be the maximum depth of move-tree examination. Suppose, for example, that we take

maxdepth = 4;

and

biggermax = {<4,6>, <6,7>};

this case, the move tree developing from a given position will normally be examined to a depth of 4 moves. However, we will search two levels below any active position found at depth 4, and then again one level below any active position encountered at depth 6, before cutting off our search at an absolute maximum depth of 7.

iv. A boolean-valued function *active(posn)* is used in the following algorithm to separate dynamic from static positions. We will use two global parameters *gain* and *minalternatives* to characterize 'dynamic' or 'active' positions. A position *posn* in which there exists a move m such that

(approxvalue(posn) + gain) le (-approxvalue(newposn(posn,m)))

is considered to be dynamic. A position *posn* in which there exist less than *minalternatives* moves m such that

(approxvalue(posn)- gain) ge (-approxvalue(newposn(posn,m)))

is also considered to be dynamic. Code for a simple routine *active(posn)* incorporating these conventions is given below.

The explanations given above should make it easy to read our final revised move-choice algorithm, which is as follows.

```
scope best; global move, maxdepth, biggermax, surveydepth,
    approxvalue, possmoves, newposn, gain;

definef bestmove(posn,maxdepth,biggermax,surveydepth);
/* third revision of game playing program */
dpth = 0; posvalue = valued(posn,Ω,Ω,dpth,maxdepth);
/* the preceding function call establishes the value of move */
return move;
end bestmove;
end best;

definef valued(posn,mymax,mymin,depth,maxdepth);
```

```
include best*;
curmaxdepth = maxdepth;
if depth eq curmaxdepth then
    /* check to see if active position calls for increase
      in current maxdepth */
    if n active(posn) or biggermax(maxdepth) eq Ω  then
      /* no increase in maxdepth */ return approxvalue(posn);;
    /* else, if the position is active and maxdepth can be increased*/
    curmaxdepth = biggermax(maxdepth);
end if depth;
/* establish depth of preliminary survey */sdepth=surveydepth(depth);
if depth ge(curmaxdepth-sdepth) or sdepth eq 0 then
    /* no preliminary survey; establish arbitrary order of moves*/

    ordermoves = [+: m ∈ possmoves(posn)]<m>;
else /* carry out preliminary survey */
    surveytupl = [+: n ∈ possmoves(posn)]<< m,valued(newposn(posn,m),
                    Ω,Ω,curmaxdepth-sdepth+1,curmaxdepth)  >>;
    /* now sort surveytupl in order of decreasing values of
      first components of entries */
    sort surveytupl;  /* the routine sort will not be shown */
    ordermoves = [+: c(n) ∈ surveytupl] <hd c>;
end if depth;
/* now make finer estimate of position value */
bestilnow = mymin;
(∀m(n) ∈ ordermoves)
    valmove = -valued(newposn(posn,m),-bestilnow,-mymax,depth+1,
                                              curmaxdepth);
    if valmove gt (mymax orm (valmove+1))
    /* since here Ω represents 'plus infinity' */
        then return valmove;;
    if valmove gt (bestilnow orm (valmove-1))
    /* since here Ω represents 'minus infinity' */
        then bestilnow = valmove;
        if depth eq 1 then /* save best move */ move = m;;
    end if valmove;
end ∀m;
return bestilnow;
end valued;
```

```
definef active(posn);
   lude best(possmoves, approxvalue, gain);
if ∃m ∈ possmoves(posn)|
    (approxvalue(posn)+gain) le (-approxvalue(newposn(posn,m)))
        then return t;
else if (#{m ∈ possmoves(posn)|
    (approxvalue(posn)-gain) ge (-approxvalue(newposn(posn,m))})
                                              lt minalternatives
    then return t;
    else return f;;
end active;
```

Still further improvements, valuable for important classes of games, can be made in these fundamental move-choosing routines. If one has an approximate value function which will very probably not misrepresent the true position-value function by more than an amount k known in advance, then moves m for which

```
(-approxvalue(newposn(posn,m))) + (2*k) lt
      [max: m∈possmoves(posn)] (-approxvalue(newposn(posn,m)))
```

need not be examined. Still better, if one has available a 'move generator' which, given a position *posn*, will produce, in order, a sequence of moves in which moves of high value will always occur early, one can call upon this generator to produce a selection of moves to explore, and can completely ignore all other elements of the set *possmoves(posn)*. However, we refrain from exploring the algorithms suggested by these reflections, and turn now to discuss artificial intelligence algorithms of another sort.

Many of the tasks addressed in artificial intelligence studies may be modeled as *graph-searches*. In a graph searching problem, one has a set N of nodes (which may be an infinite set of which a larger and larger part is constructed as the search proceeds), and for each node n, a mapping *cesor(n)* determining the (always finite) set of nodes which can be reached in one step from n.

Certain nodes n are distinguished as *goal nodes* of the search. These are distinguished by a boolean function *isgoal*, given *a priori*, such that isgoal(n) eq t if and only if n is a goal node.

A graph-search problem will often have useful properties of symmetry. Such symmetries exist if there is some group of transformations of the graph into itself which maps goal nodes into goal nodes. In the algorithms given below, such symmetry groups will be represented by a boolean function $similar(n_1,n_2)$ of two nodes, which we assume to have the value t if and only if n_1 can be mapped into n_2 by an element of the symmetry group. Note that, if similar(n_1,n_2) eq t, then any path from n_1 to an eventual goal node is matched by a corresponding path from n_2 to some other goal node. Thus, once a path from the initial point *init* to n_1 has been constructed, the existence of paths from *init* to n_2 becomes a less interesting question, which in some cases may be ignored.

On occasion, some type of *cost* will be associated with the edges of a graph to be searched; in this case, one may wish to find a path of minimum cost from an initially given node to a graph node. In the algorithms which follow, the cost of traversing an edge n_1,n_2 is represented by a two-parameter function $cost(n_1,n_2)$.

If a graph search is not to blunder in undesirably random fashion about the nodes of a graph, it must be guided by some heuristic principle allowing steps toward a goal node to be selected preferentially above other possible steps. For this reason, we assume that a function *heuristicvalue(n)*, estimating the expected minimal cost of a path from any node n to a goal node, is given *a priori*.

We shall shortly give SETL code for a general heuristic graph-search algorithm. The code is broken into three principal subroutines *heuristicsearch*, *addnewnode*, and *nextexpand*. The first of these is a master search routine, which will return a path from *init* to a goal node if it can find one, and return Ω if it definitely fails. The routine *addnewnode* handles the addition of a new node to the collection of nodes known to be reachable from *init*.

The routine *nextexpand* determines the node to be expanded next during the process of search; it uses a very simple auxiliary function *betterthan* whose details reflect the particular search strategy adopted. A tuple called *ranking* is used to keep nodes in an order allowing rapid and efficient choice of the next node to be expanded; this data object is accessed both by the *nextexpand* routine and by an auxiliary routine called *repair*. The approach employed is as follows. Nodes still to be expanded occur as components of *ranking*, their arrangement having the so-called 'heap' property (which is also exploited in the 'heapsort' procedure, cf. Item 14, Section 5B). That is, the item in position n of the *ranking* tuple is guaranteed to outrank the items in positions 2n and 2n+1. This implies that the most desirable node to expand is available in position 1 of the *ranking* tuple. Moreover, this particular 'heap' property of an arrangement is easy to maintain as new nodes are added and old nodes deleted and processed. If k nodes are present, both the insertion of a new node into proper position in the *ranking* tuple and the deletion of an old node require only log k steps.

It deserves to be remarked that the scheme explained in the preceding paragraph is of general application. It can be found useful in any situation in which numerous generated items must be retained in an order allowing the 'best' of them to be repeatedly selected for processing. Like the more general balanced tree scheme described in volume 3 of Knuth's treatise, it is considerably more efficient than either maintaining an unordered collection and searching it repeatedly for a best item, or maintaining a completely sorted collection and rearranging it with each insertion. An example of another program in which the 'heap' scheme described in the preceding paragraph or the 'balanced tree' scheme described by Knuth could have been employed is furnished by the Huffman tree building routine of section 5; if a very large Huffman tree is to be built, use of one of these devices can achieve substantial advantage.

The main *heuristicsearch* routine is as follows:

```
scope heuristic;  global nodes, rank, ranking, parent, children,
   costohere, hvalue, cost,  similar;
definef heuristicsearch(init,isgoal,cesor,similar,
                                heuristicvalue, cost);
parent = nℓ;  children = nℓ; /*node descent and ancestry mappings*/
nodes = {init}; /*total collection of nodes generated*/
ranking = <init>;  /* ordered collection of nodes generated */
/* auxiliary vector used to select best node to expand */
rank = nℓ;  rank(init) = 1;
/* rank(nd) ne Ω characterizes nodes to be processed */
costohere = nℓ;  costohere(init) = 0;
/* cost of reaching  initial node  is zero */
hvalue = nℓ;
/* mapping giving precalculated heuristic value of nodes */

(while nextexpand(ranking) is newnode ne Ω)
/* main search loop */
    if isgoal(newnode) then /* reconstruct path */
         path = <newnode>;
         (while parent(newnode) is newnode ne Ω)
              path = <newnode> + path;
         end while;
         return path;
    end if;
    /* otherwise if goal not reached */
    (∀n ∈ cesor(newnode)) addnewnode(n,newnode);;
end while;
/* if no more nodes can be constructed then */ return Ω;
end heuristicsearch;
end heuristic;

define  addnewnode(child,parentnd);
include heuristic(cost,similar,children,costohere,nodes,rank,
                                                    ranking);
costochild = costohere(parentnd) + cost(parentnd,child);
/* now check to see if child has already been encountered */
```

```
   if child n ∈ nodes then go to  newchild;;
      costochild lt costohere(child) then
       /* we have found a lower cost path to child. we therefore
           modify its recorded cost and parentage */
       oldparent = parent(child);
       children(oldparent) = children(oldparent) less child;
       install(child,parentnd,costochild);
       if rank(child) is  rnk ne Ω then
            repair(ranking,rnk,f); /* f signals no deletion */
            /* now recursively reprocess the descendants of  child*/
            (∀n ∈ children(child) orm nℓ) addnewnode(n,child);;
       end if rank;
   end if costochild;
   return; /* note that new paths to old nodes will be ignored
       unless they have cost advantages */
newchild: /* the child node is new. test it for similarity
                with an old node */
       if ∃n ∈ nodes | similar(n,child) then
            /* drop more 'costly' of child and n */
            if costochild lt costohere(n) then
                 nodes = nodes with child;
                 oldparent = parent(n);
                 children(oldparent) = children(oldparent) less n;
                 install(child,parentnd,costochild);
                 ranking(#ranking+1) = child;
                 repair( #ranking, f);        /* f signals no deletion*/
                 /* remove n and  its descendants */
                 removedescs(n);
            end if costochild;
       else /* case in which new node is not similar to any
            existing node */
            install(child,parentnd,costochild);
            nodes = nodes with child;
            ranking(#ranking+1) = child;
            repair( #ranking, f);        /* f signals no deletion */
```

```
    end if ∃;
    return;/* note that nodes similar to old nodes will be ignor
         unless paths to them having cost advantages are found */
    end addnewnode;

define  install(child,parentnd,costchild );
/* auxiliary  routine establishing child as a descendant of parentnd*/
include heuristic(costohere, parent, children);
costohere(child) = costchild;
parent(child) = parentnd;
children(parentnd) = children(parentnd) orm nℓ with child;
return;
end install;

define  removedescs(parentnd);
/* auxiliary routine removing the descendants of a given node,
    unto the uttermost generation */
include heuristic(children,parent,nodes,rank,ranking,
                                          costohere);
(∀n ∈ children(parentnd) orm nℓ)
    removedescs(n);
end ∀n;
parent(parentnd) = Ω;
children(parentnd) = Ω;
costohere(parentnd) = Ω;
nodes = nodes less parentnd;
if(rank(parentnd) is rnk ne Ω) then repair(rnk,t);;
    /* t signals deletion */
return;
end;

definef nextexpand(ranking);
/* return first-ranked item and repair ranking */
keep = ranking(1);
if keep eq Ω then return Ω;;
repair(1, t);          /* t signals deletion */
return keep;
end nextexpand;
```

```
definef repair(n,deleteflag);
   routine to rearrange the items in the ranking vector after the
    n-th item is removed, while preserving the 'heap' property
    of this vector */
/* if deleteflag is true, position n in the ranking vector
    will be deleted */
include heuristic(rank,ranking);
if n deleteflag then go to arrange;;
/* deleted element becomes unranked; remove it from rank */
rank(ranking(n)) = Ω;

/* replace deleted component by last component of ranking */

ranking(n) = ranking(#ranking); ranking(#ranking) = Ω;
/* now proceed to restore the 'heap' property of the ranking vector*/
arrange:   posn = n; dval = ranking(n);
           flow              dvalgtparent?
             interwithparent+          dvalltonedesc?
             dvalgtparent,       interwithbiggerdesc+  quit,
                                   dvalgtparent;
dvalgtparent:= if posn eq 1 then f else
                          betterthan(ranking(posn),ranking(posn/2));
interwithparent: oldposn = posn;
          <ranking(posn), ranking(posn/2),posn>
               = <ranking(posn/2),ranking(posn),posn/2>;
          rank(ranking(oldposn))  = oldposn;
dvalltonedesc:=if (2*posn) gt #ranking then f
             else if(2*posn+1) gt #ranking then
                          betterthan(ranking(2*posn),dval)
             else betterthan(ranking(2*posn),dval)
               or betterthan(ranking(2*posn+1),dval);
interwithbiggerdesc: bposn = if(2*posn+1) gt #ranking then 2*posn
             else if betterthan(ranking(2*posn),ranking(2*posn+1))
               then 2*posn else 2*posn+1;
          oldposn = posn;
          <ranking(posn),ranking(bposn),posn> =
                <ranking(bposn),ranking(posn),bposn>;
          rank(ranking(oldposn)) = oldposn;
          end flow;
```

```
        /* now note the rank to which dval has moved */
rank(dval) = posn;
return;
end repair;
```

The order in which nodes are chosen for expansion by the above algorithm depends on the precise form of the *betterthan* function called by the *repair* routine. By varying the form of this function, we vary the heuristic controlling our pattern of search. The following versions of the *betterthan* routine are often used:

```
definef betterthan(n1,n2);
include heuristic(hvalue,costohere);
return(hvalue(n1) + costohere(n1)) lt (hvalue(n2)+costohere(n2));
end betterthan;
```

This form of the *betterthan* function gives a standard heuristic search, called the A*-algorithm in Nilsson's well known book on artificial intelligence. (Often the *cost* function is given simply by cost(n1,n2) ≡ 1.) The 'equal cost search' (which, if the cost function is constant, becomes the so-called breadth-first search) is obtained by taking heuristicvalue(n1) ≡ 0, i.e., by dropping the *hvalue* terms out of the *betterthan* routine shown above.

If a graph to be searched is truly maze-like, i.e., if no *heuristicvalue* or similarity functions substantially aiding the search for a goal node is available, then the search algorithm given above reduces to a transitive closure procedure which essentially adds nodes and paths at random until a goal node happens to be found. Assuming for simplicity that the graph being searched is connected, and that it contains n nodes, of which a smaller number k are goal nodes, we must expect a collection of nodes roughly numbering n/k to be examined before a goal node is found. In such situations backwards-and-forwards searching, of which we shall now describe a special case, can have substantial advantage. Suppose, to be specific, that we are given a graph,

- starting node, and a single target node, and that our problem to find a path connecting the starting node to the target node. We then simultaneously construct paths backward from the target node and forwards from the starting node, looking for an intersection. As our search proceeds, we keep the group F of nodes reached along forward paths roughly equal in number to the group B of nodes reached along backward paths.

The advantage of this tactic of search is shown by the following rough calculations. Let the graph being investigated consist of n nodes, and suppose that k_1 nodes of group F and k_2 nodes of group B have been constructed. Assuming that $k_1, k_2 << n$, the probability that some one of the nodes of group B also belongs to group F (thereby completing a path) is roughly $k_2 \cdot (k_1/n)$. The probability becomes substantial (say, $\geq 1/2$) when $2k_1k_2 \geq n$. With k_1 roughly equal to k_2, a substantial probability that F and B intersect arises when roughly $\sqrt{n}$ nodes have been added to F ∪ B. If we use forward searching only, i.e., keep k_2 fixed at 1, then n/2 nodes must be added to F before the probability that F intersects B becomes as large.

In the following forward-backward path-search algorithm, which is of the type suggested by the preceding considerations, the graph to be searched is represented by a function *cesor(n)* defining the set of all nodes which can be reached in a single step from a given node n; we also suppose that a map *pred(n)*, defining the set of all nodes m from which n can be reached in a single step, is given.

```
definef fbpathsearch(cesor,pred,init,targ);
/* construction of path by forward and backward search */

fset = {init}; bset = {targ};
parent = nl; /* parentage of nodes in fset and bset */
fnew = fset; bnew = bset;
search:   if (#fset) gt #bset then go to growb;;
          nd from fnew;
          newer = cesor(nd) - fset;
```

```
        if ∃n ∈ newer|n ∈ bset  then /* path has been constructed*/
             fend = nd;  bend = n;
             go to buildpath;
        end if;
        fnew = fnew+newer;  fset= fset + newer;
        (∀n ∈ newer) parent(n) = nd;;
        if fnew eq nl /*so that no path exists*/ then return Ω;;
        go to search;
growb:  nd from bnew;
        newer = pred(nd) - fset;
        if ∃m ∈ newer|m ∈ fset then /*path has been constructed*/
             fend = m;  bend = nd;
             go to buildpath;
        end if;
        bnew = bnew + newer; bset = bset+newer;
        (∀n ∈ newer) parent(n) = nd;;
        if bnew eq nl /  so that no path exists */ then return Ω;;
        go to search;
buildpath: path = nult;
        (while fend ne Ω doing fend = parent(fend);)
             path = <fend> + path;
        end while;
        (while bend ne Ω doing bend = parent(bend);)
             path = path + <bend>;
        end while;
        return path;
        end fbpathsearch;
```

Many other parts of the somewhat amorphous domain of artificial intelligence have been sources of interesting algorithms. We shall give only one more algorithm belonging to this subject; before doing so, however, we survey some of the main problems of this area, from which important algorithms may be expected to arise in the future. These subareas are:

i. Gestalt matching: This is the problem of creating algorithms which will duplicate certain aspects of the organic brain's sensory processing, well enough at least to enable

computer recognition of artifacts (such as written letters and spoken sounds) which are regularly used for communication between persons. Note that the problem here is partly psychological; we want a computer to recognize the same characteristics of written letters that *we* recognize, so that we can use introspection as an aid in comprehending its reactions, and so that we can communicate with it in a manner natural for *us*. There is good evidence that the functions to be duplicated are, to a large extent, innate rather than learned; thus it is reasonable to seek for algorithms with a relatively fixed core, though enough flexibility to permit a considerable measure of subsequent learning is undoubtedly desirable.

ii. Feature seeing: In both the sensory and the intellectual areas, coping with complex and varying situations will require the ability to discern key features within complex logical patterns. To make it convenient to write programs with this ability, and to leave open the possibility of easy growth in the set of features recognized vital both for externally directed experimentation and for self-regulating learning, languages different from the ordinary serial procedural kind may be appropriate. Such languages might perhaps be closer to present-day simulation languages than to procedural languages, and might for example specify that whenever certain combinations of conditions prevail, an abstract node representing this fact is to be created, which node can then function as an input for the creation of additional nodes. Certain of the parsing algorithms we have already studied, especially those for parsing by the method of nodal spans, are suggestive in this connection.

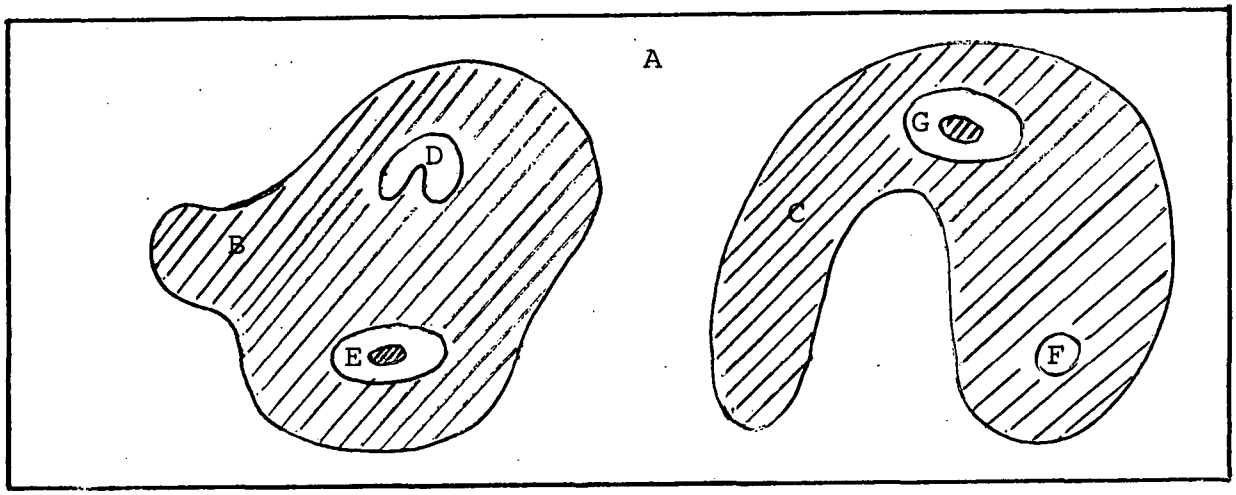
iii. Learning: Many of the problems with which one would like to be able to deal involve interaction with enormous numbers of subcases which cannot be searched exhaustively. One's only chance of success with problems of this type will lie in the development of effective processes of *sporadic search*. Such search processes will be guided by a set of 'features noticed', and by an evaluation of the significance of these features. The more finely tuned the set of features and the more just their evaluation, the more effective a sporadic search guided by these features will be. While

sets of features and weights for them can initially be elicited by detailed interviews with experts in a particular field to be treated, a 'hand' method of this kind will eventually break down. When it does, it will become necessary for programs to experiment independently in the fields to which they are dedicated. Such programs must both modify their evaluations of old features and elaborate new features, in whatever manner is indicated by the outcome of automatically conducted experiments. This is the process of *computer learning*. What counts here is first of all *efficiency of learning*, i.e., the size of the computation needed to attain a certain degree of progress. Methods which extract as much useful information as possible from an experiment of given cost are needed; present machine learning techniques are highly deficient in this regard. Moreover, as a learning process leads to the discovery of new features, internally maintained auxiliary data sets will tend to grow. Methods for controlling this data-base growth, for eliminating old features subsumed by more powerful new features, and for ensuring that the whole process of feature-seeing, evaluation, and discovery remains reasonably efficient, are required. This is a mechanical version of the familiar scientific process of "literature pruning."

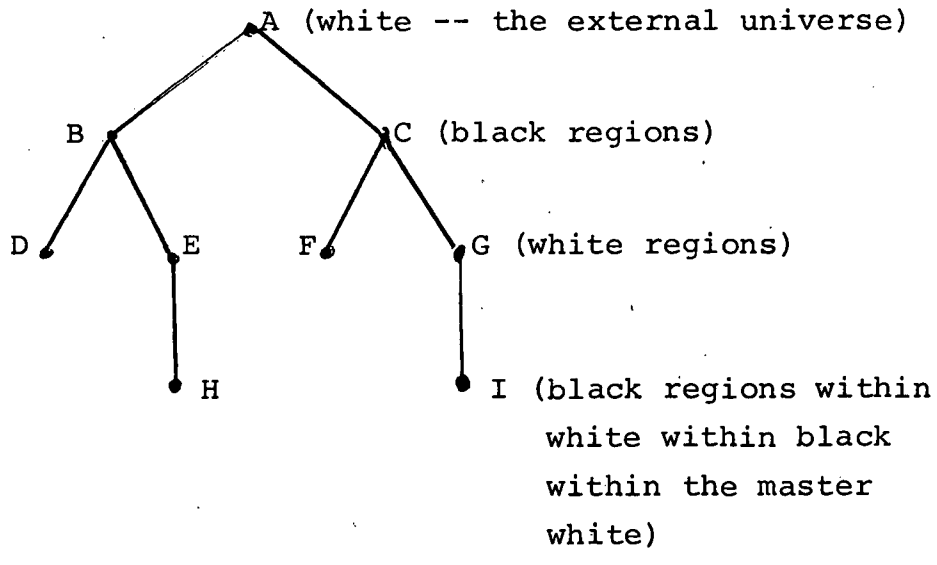
iv. Generalization: We have noted that, as a mechanical learning process unfolds, it will occasionally become necessary to compress the mass of information concerning special features and situations which it builds up. The best way of doing this is to replace large tables by small efficient programs capable of calculating all of or most of their entries. The iterative application of such 'condensing optimizations' is the process of *mechanical generalization*. This observation reminds us that, just as the proper study of mankind is man, so the proper study for artificial intelligences is computer science; intelligent programs ought to devote as much effort to studying their own performance and discovering ways to improve it as they devote to their originally assigned prime purposes.

Having relieved ourselves of these ex cathedra remarks, we go on to the detailed business at hand. Our final algorithm,

...e to Buneman, belongs to the subarea of gestalt finding, ...d more specifically to visual pattern processing. It aims to uncover important topological relationships (connectivity and insideness-outsideness) implicit in plane figures. Given a plane figure made up of black and of white regions, it produces an abstract *connectivity graph*, each of whose nodes represents a connected component (either black or white) of the figure. A node representing a component R is a descendant of a node representing a component R' if and only if R is inside R'. For example, if the analysis procedure to be given is applied to the figure

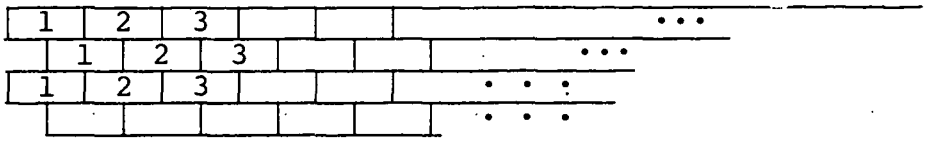


it will produce the following graph.

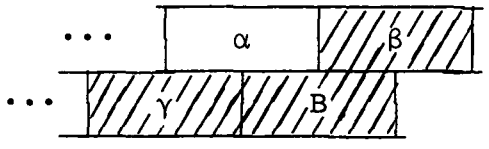


This graph tells us these facts: within an outermost white region (the 'background', represented by the root node of our tree), the are two connected but mutually disconnected (black) subregions. Each of these encloses precisely two (white) subregions; in each case, one of these two white subregions encloses a (black) region.

The topological analysis algorithm given below assumes that the plane area which it will analyze is decomposed into rectangles forming a brick-like pattern of the following sort.



We assume that all the bricks constituting the outer boundary of the pattern are white. Each brick B is in contact with six other bricks, two in the row above B, and a like number in the row containing B and in the row below B. We suppose each brick to be designated as being either 'white' or 'black'. Our algorithm works over the bricks of an area to be analyzed, proceeding down through successive rows, and within each row in left to right order. The first or 'prime' brick belonging to a connected region R of given color is used to symbolize R. Each subsequent brick B found to touch a brick B' of R having the same color as B is assigned to the same region. This is done by use of an auxiliary map *represent*, and by executing the statement represent(B) = B' . Given this mapping, the region R containing B is determined by calculating represent(represent(...represent(B)...)), iterating until a 'prime' brick B", distinguished by the fact that represent(B") eq Ω, is encountered. This prime brick symbolizes the region R. A brick B occurring in the configuration



serves either to unite the distinct like-colored regions to which the bricks labeled β and γ belong, or, if these regions

are already the same, serves to complete an enclosure whose inside the region containing the differently colored brick labeled α. The first of these two cases is characterized by the fact that the prime brick $\bar{B}$ symbolizing the region R_β containing β is distinct from the prime brick $\bar{C}$ symbolizing the region R_γ containing γ. The second case is characterized by $\bar{B}$ eq $\bar{C}$. In the first case, we merely assign $\bar{C}$ as the representative of $\bar{B}$, thus dropping $\bar{B}$ from prime brick status and identifying R_β as identical with R_γ. In the second case, we note that the region R_α containing α is inside R_β.

In the following SETL code, a function *truerep* is used to calculate the region containing a given element α (more precisely, to calculate the prime brick symbolizing this region).

```
definef topanalyze(color,nrows,ncols);
/* nrows is the number of rows in the array of rectangles to
    be analyzed; ncols is the number of columns */
/* color(<m,n>) is t if the n-th brick in the m-th row is black */
/* represent is assumed to be global */
represent = nl;
/* represent(<m,n>) is  a prior  brick belonging to the region
    to which the brick <m,n> belongs */
/* since all the bricks in the first row belong to
    the 'background', we have: */
(1 ≤ ∀nc ≤ ncols) represent(<1,nc>) = <1,1>;;
poffset = 0; /* flag distinguishing odd and even rows. processing
    starts with the second row */
(2 ≤ ∀nr ≤ nrows)
    represent(<nr,1>) = <1,1>; /*first brick in row belongs
                                    to'background'*/
    (2 ≤ ∀nc ≤ ncols)
        /* get 3 preceding neighbor blocks */
        pnaybs = <<nr-1,nc-poffset>,<nr-1,nc-poffset+1 min ncols>,
                                    <nr, nc-1>>;
```

```
        if n ∃ nayb(n) ∈ pnaybs|color(nayb) eq color(<nr,nc>) then
            /* <nr,nc> may represent a new region */ continue;
        else represent(<nr,nc>) = nayb;
            if n eq 2 and color(pnayb(3)) eq color(pnayb(2)) then
                    /* either two regions must be identified
                            or 'surround' is complete */
                if truerep(nayb) ne truerep(pnayb(3)) then
                    represent(pnayb(3)) = nayb;
                else
                    <nayb, pnayb(1)> in inside;
                end if truerep;
            end if n eq 2;
        end if n ∃;
    end ∀nc;
    poffset = 1 - poffset;  /* reverse 'odd row' flag */
end ∀nr;
/* now modify the 'inside' map, basing it upon the 'prime'
    nodes of each of the regions */
return {<truerep(x(1)),truerep(x(2))>, x ∈ inside};
end topanalyze;

definef truerep(brick);
b = brick;
(while represent(b) ne Ω)
    b = represent(b);
end while;
return b;
end truerep;
```

At the cost of some complication of its detail, the routine *topanalyze* shown above can be modified so as only to require that values of *represent* and *color* for a single row of a total array of bricks be stored. We leave consideration of this improvement to the reader.

TEM 16. Some Optimization Algorithms.*

1. Graph ordering.

Many interesting algorithms for the optimization of compiler-generated code are based on the analysis of an abstract graph representing the flow structure of the program. This graph, the so-called *program graph* of the program, is defined as follows.

Call any sequence of instructions which has the property that control always enters the sequence at its first instruction and always leaves the sequence at its last instruction a *basic block*. If the last instruction in one basic block b_1 might transfer control to the first instruction in another block b_2, call b_2 a *successor block* of b_1. The nodes of a program graph are then the basic blocks of a program, and the successors of a node its successor blocks. The node containing the first (entry) instruction of a program we call the *entry node* e; any node terminating in an "exit" instruction is considered to have no successors, and is called an *exit node*. We suppose that every node of a program graph is reachable from its entry node through a chain of successors; any node not reachable in this way represents code that can never be executed and may as well be deleted from the program. In what follows, any sequence of program-graph nodes in which the n-th is always a successor of the (n-1)st will be called a *path*.

Various graph-theoretical notions related to the program graph of a program are useful in analyzing the flow of control and data relationships during program execution. The first of these relationships which we shall consider is that of *predomination*. A node b is said to *predominate* a node c if every path from the entry node to c must pass through b. The predominators of c are useful in various ways in optimization; for example, under certain conditions, code can be moved from c to certain of its predominators.

---

* This section contains some material drawn from the earlier manuscript *Abstract Algorithms*; it also draws upon material due originally to K. Kennedy, P. Owens, and others, which appeared in SETL Newsletters 28, 37, and 38. Many of the algorithms which follow have been checked against a library of debugged SETLB optimization algorithms being developed by D. Shields.

Here is an algorithm, due to F. Allen, for finding all the predominators of all the nodes in a program graph. First observe that if, starting from the entry node, we can reach (find a path to) a node c without passing through a node x, then we can reach any immediate successor y of c without passing through x, except, of course, if y is x. Designate the set of nodes x which can be reached without passing through c as *notfor(c)*. Thus the set of predominators of c consists of all the nodes not contained in *notfor(c)*. Here is the SETL algorithm. (Note the use of the function a orm b , which returns 'if a ne Ω then a else b'.)

```
definef predoms(nodes,entry);

/* the successor function cesor is assumed to be global */
/* 'nodes' is the set of program nodes, 'entry' is the entry node,
   'cesor(x)' is the set of all successors of node x. This function
   returns the set 'dom' such that dom(x) is the set of all pre-
   dominators of x */
/* initially, no nodes are needed to reach entry */
notfor = nl;; notfor(entry) = nodes less entry; todo = {entry};
(while todo ne nl)
   c from todo;
   (∀y ∈ cesor(c)) /* update set of elements which are not
      predominators of y, put y back on workpile if this set changes*
      new = notfor(c) -(notfor(y) orm nl) less y;
      if new eq nl then continue;;
      /* else */ notfor(y) = notfor(y) orm nl + new;
      y in todo; /* since the successors of y must be processed */
   end ∀y;
end while;
dom  = nl;
(∀y ∈ nodes) dom(y) = nodes - notfor(y) less y;;
return dom;
end predoms;
```

Next we give an algorithm, due to Earnest, Balke, and Anderson, for linearizing a program graph in an advantageous order. This order is, among other remarks that may be made concerning it, an order in which all the predominators of a node precede the node. The algorithm is as follows:

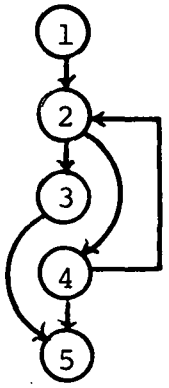
i. starting at the entry node, and always without repeating y nodes, generate a path.

ii. when this path can no longer be extended, back up along it to a node x from which a new node of the graph is seen as a successor. Starting at this new node, generate a path; and insert it in linear order into the old path, immediately after x. Continue until the whole graph is linearized.

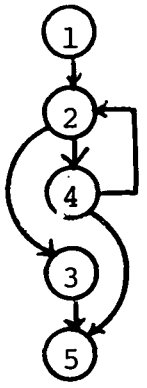
In SETL, we have:

```
definef graphord(nodes,entry);
/* the successor map cesor is assumed to be global */
order = <entry>;
mark =  {<entry,t>};
jlast = 1 /* jlast is highest numbered node from which new path
   may begin */;
(while jlast ≥ ∃j ≥ 1, last ∈ cesor(order(j))|mark(last) ne t)
/* start new path */
   path = <last>;  mark(last) = t;
/* and extend as far as possible */
   (while ∃ next ∈ cesor(last)|mark(next) ne t)
        path = path + <next>;
        mark(next) = t; last = next;
   end while ∃;
/* insert path after jth node in order */
   order = order(1:j) + path + order(j+1:);
   jlast = j + #path + 1;
/* note path(#path) has no unmarked successors */
end while jlast;
return order;
end graphord;
```

A program graph straightened by the preceding algorithm is said to be in *straight order*. But in this order it may still contain configurations, such as the following,



which it is advantageous to rearrange as



This may be accomplished by applying a *loop-cleansing* procedure, also due to Earnest, Balke, and Anderson, which may be shown to preserve the principal properties of the ordering established by the preceding procedure. The loop-cleansing procedure is as follows.

i. Perform step (ii) for each node n, procedures from last to to first (i.e. in reverse straight order).

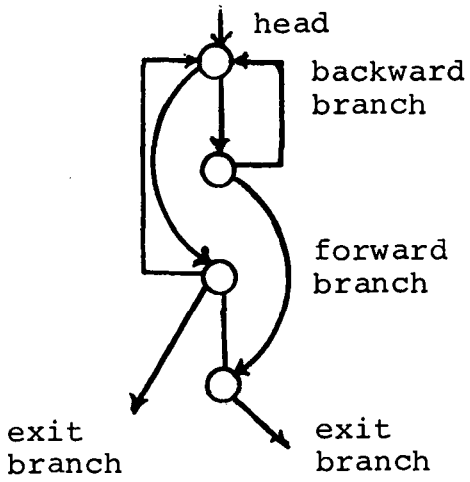
ii. Find all nodes following n from which n may be reached by a backward branch; we call these nodes *latches*. Proceed through the set of latches of n in straight order, performing the following for each latch m: Mark m, its predecessors, their predecessors, etc.; call the last node marked in this way ℓ. Then push all the marked nodes up-toward n and all the unmarked nodes down toward ℓ, keeping the marked nodes in their established order, and the unmarked nodes in theirs.

Note the use in the following code of the transitive closure function *closure* introduced on page 119.

```
/* generate predecessor map */
  ed = {<n, nl>, n ∈ nodes};
(∀x ∈ nodes, y ∈ cesor(x)) pred(y) = pred(y) with x;;
/* we assume that order(i) is the i-th node in straight
   order, as returned by the graphord function described above */
/* generate node number */
number = {<order(i),i>, 1 ≤ i ≤ #nodes};
(#nodes ≥ ∀n ≥ 1)
   head = order(n);
   latches = {nod ∈ pred(head)| number(nod) gt n};
   k = n+1;
/* nodes with indices at least k are considered to be 'unmarked' */
   (while latches ne nl) /* find element of minimum index */
        mn = [min: nod ∈ latches] number(nod);
        m = order(mn);
        marked = closure(pred,{m},{order(j), k≤j≤#nodes});
        latches =  latches - marked;
/* since nodes once processed need never be processed again */
/* shift unmarked elements down, others up */
        (while k ≤ ∃j < #nodes | n order(j) ∈ marked
                  and order(j+1) ∈ marked)
             <number(order(j)), number(order(j+1)),order(j),order(j+1)>
             = <j+1, j, order(j+1),order(j)>;
        end while k;
        k = k + #marked;
   end while latches;
end ∀n;
```

2. Intervals, derived graphs, and live-dead analysis.

Next we turn to study certain notions, useful in the analysis of program graphs, due to J. Cocke and F. Allen. An *interval* in a program graph is a set s of nodes, containing a distinguished node x called the *head* of s, such that there is no entry into s except through x, and such that when x is removed, s is free of ps (a *loop* is a closed path in a program graph). The following ure shows a typical interval.



Single entry regions of this kind serve to make specific the heuristic notion of "inner loop". They are very useful for program flow analysis, and systematic optimization procedures can be built upon them.

It is a characteristic property of intervals that their nodes can be enumerated in such a way that, with the exception of branches terminating at the interval head, all branches between nodes of the interval are "forward" branches, i.e., go from a node x to a node y having a larger serial number in the enumeration of the interval.

The *interval of a node* x is the largest interval with x as head; it may consist of x only.

Here is an algorithm which determines the interval of a node x, and enumerates the nodes of this set in an order having the property noted above. It also finds the set of all successors of nodes of the interval which do not belong to the interval; this will be needed below.

```
definef interval(nodes,x);
/* npreds, followers and cesor  are assumed to be global */
/* count the number of predecessors of every node */
npreds = {<x,0>, x ∈ nodes};
(∀x ∈ nodes, y ∈ cesor(x))
   npreds(y) = npreds(y) + 1;;
int =nult; followers = {x}; count = {<y,0>, y ∈ nodes};
count(x) = npreds(x);
/* 'count' will be a count of the number of predecessors of
   a node which belong  to the interval being constructed */
```

```
while {y ∈ followers|npreds(y) eq count(y)} is newin ne nl)
   (∀z ∈ newin)
        int(#int+1) = z;
        z out followers;
        (∀y ∈ cesor(z)|y ne x) count(y) = count(y)+1; y in followers;;
   end ∀z;
end while;
return int;
end interval;
```

An interval is called *maximal* if it is not contained in any larger interval. It can be shown that every program graph can be decomposed uniquely into a union of maximal intervals, and that distinct maximal intervals are disjoint. This decomposition, the so-called *Cocke-Allen decomposition* of a program graph, is quite useful in flow analysis.

To find these maximal intervals, we proceed as follows. Take the interval generated by the entry node of the program graph; this is a first maximal interval. Then, iteratively having formed other maximal intervals, take any successor x of a point in these intervals not lying in any of them, and form the interval of x; this is a new maximal interval.

The following SETL algorithm realizes this process. It also associates, with each maximal interval, the set *follow(int)* of all nodes which are successors of a node of the interval without belonging to the interval; and with each node b of the program graph the maximal interval *intov(b)* which contains it.

```
definef intervals(nodes,entry);
/*          followers, follow, intov  are all assumed to be global*/
ints = nl;  seen = {entry};  follow = nl; intov = nl;
(while seen ne nl)
   node from seen;
   interval(nodes,node) is i in ints;
   follow(i) = followers;
   (1 ≤ ∀k ≤ #i) intov(i(k)) = i;;
   seen = seen +  {x∈followers|intov(x) eq Ω};
end while;
```

```
return ints;
end intervals;
```

The *derived graph* g' of a program graph g is defined as follows: the nodes of g' are the intervals of g; the successors of an interval *int* are the intervals distinct from *int* containing successors of the nodes within *int*; the entry node of g' is the interval containing the entry node of g. The derived graph of a program graph gives a simpler, "coarsened" representation of its program flow; in this representation "inner loops", i.e., intervals, appear as single points, and only "outer" loops are shown. If g contains any interval of more than one point, g' will have fewer nodes than g. A program graph in which no interval of more than one point can be found is called an *irreducible graph*; fortunately, such graphs arise only rarely in connection with actual programs. In SETL, we may write the criterion of irreducibility very simply as follows:

```
(#nodes) eq #intervals(nodes,entry) .
```

The definition of the derived graph in SETL is also quite trivial; here it is:

```
definef dg(nodes,entry);
   /* cesor, follow, intov, dent, pred are all assumed to be global*/
ints = intervals(nodes,entry);  dent = intov(entry);
(∀i ∈ ints) cesor(i) = intov[follow(i)];;
(∀i ∈ ints) pred(i) = {int∈ints|i∈cesor(int)};;
return ints;
end dg;
```

By forming successive derived graphs g', (g')', etc. of an original graph g, we look at an original program in a more and more global way. In cases in which this *derivation sequence* converges to a graph consisting only of a single node, we may claim that the method of intervals provides a decisive analysis of program flow. Graphs having this property are called *eventually reducible* program graphs.

Here is the SETL definition of the derived sequence of a grapn.

```
dfinef dseq(nodes,entry);  /* dent is 'global' */
  q =    <<nodes, entry>>;  <n,e> = <nodes,entry>;
(while #(dg(n,e) is der) lt #n doing <n,e> = <der,dent>;)
   seq(#seq+1) = <der,dent>;;
return seq;
end dseq;
```

Intervals have many uses in the optimization-analysis of programs. Generally speaking, the derivation sequence of a program gives a very useful guide to the order in which various optimizing processes can most effectively be applied. We shall now discuss one such application; more specifically, we shall study the so-called "live-dead" analysis of variables.

A variable x is said to be *live* at a given point q in a program if from this point there exists a path P in the program, not passing through any instruction which assigns a value to x, and such that P terminates in an instruction which uses the value of x. In the contrary case, x is said to be *dead* at q. It is clear that if x is live at given point in a program, its value must be saved in some known register or core location for later use. Conversely, when x becomes dead, the register containing it becomes available for other use.

We shall now describe an algorithm, due to K. Kennedy, for efficiently deriving live-dead information for any eventually reducible program graph. Of course, to carry out such an analysis, we require that basic information concerning the use of variables within a program be available. We suppose that this information is made available according to the following conventions.

i. All the variables with which the program is concerned are collected into a set of *vars*;

ii. With each basic block b is associated a set *blkuses(b)* consisting of all the variables x in the set *vars* such that, starting at the entrance to b, a use of x within *b* will be reached before any assignment is made to x.

iii. With each basic block b is associated a set *thru(b)*, consisting of all the variables x in the set *vars* such that b contains no operation implying an assignment to x.

A path through the program which does not encounter any instruction assigning a value to a variable x is called an x-*clear* path

We shall wish to associate certain sets with each interval *intv*. These are as follows:

1. The set of all x such that there is an x-clear path in *intv*, starting at the entrance to *intv* (i.e., immediately before any instruction in *intv* is executed) and terminating at a use of x within *intv*.

2. For each block e of *follow(intv)*, the set of all x such that there is an x-clear path through *intv*, starting at its entrance, and terminating at e.

We shall call the first of these sets *blkuses(intv)* and the second *thru(intv,e)*. The following observation allows these sets to be calculated easily. Let h be the head of *intv* and let b be a block of *intv*. If the blocks of *intv* are enumerated in the order defined by the basic interval construction function *interval(x)* described above, then, as has been remarked, all backward branches in *intv* lead to h. Thus all irredundant paths in *intv* from h to b i.e., all paths not containing pointless repetitions, will involve forward branches only. Therefore the information we require can be developed by a simple scheme which propagates uses backwards and which considers forward branches only. In SETL, the algorithm we require is as follows:

```
uaux = nl; taux = nl; head = intv(1);
(#intv ≥∀n ≥ 1) b = intv(n);
   forward = {y ∈ cesor(b)|intov(y) eq intv and y ne head};
   uaux(b)=blkuses(b)+ (thru(b)*([+:y∈forward] uaux(y)orm nl));
   (∀intx ∈ cesor(intv))
        if intx(1) ∈ cesor(b) then taux(b,intx) = thru(b);
        else taux(b,intx)=thru(b)*([+:y∈forward]taux(y,intx)orm nl);
        end if;
   end ∀intx;
end ∀n;
blkuses(intv) = uaux(head);
(∀intx ∈ cesor(intv))
   thru(intv,intx) = taux(head,intx);;
```

We may now observe that blkuses(intv) and thru(intv,inta) are ated to the interval *intv* in just the same way that blkuses(b) ana thru(b) are related to a basic block b. Thus the construction shown above can be repeated for the derived graph, and therefore, iterating, for all the graphs of the derivation sequence. We write the process that builds this information for all the graphs of the derivation sequence as a subroutine; except for the slight differences occasioned by the fact that intervals may have several exits while a basic block can have only one, we can use almost precisely the code which appears above. With the necessary corrections, we have the following algorithm. (Note the use of the auxiliary function a orm b which returns the value 'if a ne Ω then a else b.)

```
define builda(nodes,entry);
/* cesor,intv,blkuses,thru, seqd are assumed to be global */
seqd = dseq(nodes,entry);
(1 < ∀k ≤ #seqd, intv ∈ hd seqd(k))
  uaux=nℓ;  taux=nℓ: head=intv(1);
  (#intv ≥ ∀n ≥ 1) b = intv(n);
    forward={y∈cesor(b)|intov(y) eq intv and y ne head};
    flow                    (k gt 2)?
                        intcas,  blockcas;
   <blockcas:>uaux(b) = blkuses(b) + (thru(b)
              * ([+: y ∈ forward] uaux(y) orm nℓ));
   (∀intx ∈ cesor(intv))
      if intx(1) ∈ cesor(b) then taux(b,intx) = thru(b);
      else taux(b,intx)=thru(b)*([+:y∈forward] taux(y,intx)orm nℓ);
      end if;
   end ∀intx;
   <intcas:>   uaux(b) = blkuses(b) +( [+: y ∈ forward](thru(b,y)
             *  uaux(y))orm nℓ);
   (∀intx ∈ cesor(intv))
     taux(b,intx) =
     (if intx(1) ∈ cesor(b) then thru(b,intx(1)) else nℓ)
       +( [+: y∈forward](thru(b,y) *  taux(y,intx)) orm nℓ );
   end ∀intx;
   end flow;
end ∀n;
```

```
   blkuses(intv) = uaux(head);
   (∀intx ∈ cesor(intv))thru(intv,intx)= taux(head,intx);;
end ∀k;
return;
end builda;
```

We now define a new mapping *uses(intv)* for each basic block and interval of a program. Specifically, *uses(intv)* is the set of all variables v for which there exists a path in the program graph which starts at the head of *intv* and reaches a use of v without ever passing through an assignment to v. Our calculation of *uses(intv)* is based upon the following observation. When the blocks b of an interval *intv* are enumerated in the order we have been considering, every backward branch passes through the head of *intv*. It follows therefore that a minimum length path from b through *intv*, to either a use of a variable or to an exit from *intv*, will either consist of forward branches exclusively or will contain one and only one backward branch. Next, suppose that we are dealing with a program graph g that is eventually reducible. Then the graph g_{n-1} appearing at the next-to-last stage in g's derivation sequence consists of a single interval *without any exits*; hence the observation that we have just made gives an easy way of computing uses(b) for each block b of g_{n-1}. But the blocks of g_{n-1} correspond exactly to those blocks of g_{n-2} which are interval heads; hence, working backwards iteratively, we can compute *uses(b)* for each block of $g_{n-2}, g_{n-3}, \ldots$, until g is reached. We will then know, for each basic block b in our original program, whether or not there exists an x-clear path from the entrance of b reaching a use of x; and this evidently tells us whether or not x is live at the entrance to b.

We may write the routine which completes the construction of *uses* in SETL as follows:

```
define builduse(nodes,entry);
/* cesor, intov, blkuses, uses, thru, seqd are assumed to be global*/
builda(nodes,entry);
(#seqd ≥ ∀k > 1,  intv ∈ hd seqd(k))
   (#intv ≥ ∀n ≥ 1) b = intv(n);
      backorexit={c∈cesor(b)| intov(c) ne intv or c eq intv(1)};
      uses(b) = blkuses(b) +
         (if k eq 2 then thru(b)
            *([+: c∈backorexit] uses(intov(c)) orm nl)
         else [+: c ∈ backorexit] (thru(b,c)
            * uses(intov(c))) orm nl;
```

```
   end ∀n;
end ∀k;
  turn;
  d builduse;
```

3. An algorithm for use-definition chaining.

Various important optimizations depend on knowing which definitions in a program can affect the values of variables used at a given point in the program's control flow.

As a further example of the applications of the interval method, we now present an algorithm for carrying out a 'use-definition chaining' process.

Given a program flow graph, our algorithm will associate with each block b the set *reaches(b)* of all definitions which can reach b, i.e. the set of all definitions D for which there exists a path, clear of all redefinitions of the variable defined by D, to the block b. The algorithm consists of two phases. In the first phase, beginning with information concerning the operations occurring in basic program blocks, we collect interval-related information, processing each derived graph of an originally given flow-graph in sequence. In the second phase, once necessary global information has been gathered, we proceed in reverse order through the sequence of derived graphs, ultimately depositing appropriate live-dead information at the entry to every block.

The following functions and sets are provided as input to our analysis:

1.	*defs*	the set of all definitions in the program.
2.	*var(defn)*	the variable defined by the definition *defn*
3.	*varsthru(b,sb)*	the set of variables for which there is a definition-clear path through b to sb
4.	*defsfrom(b,sb)*	the set of definitions D within b for which there is a path to an exit to sb from b clear of redefinitions of the variable defined by D.
5.	*initial*	the set of initial definitions, made before program entry (e.g., definitions made by DATA statements in FORTRAN).
6.	*pred, cesor*	precessor and successor maps defining the structure of the program graph being analyzed.

Note also that *nodesof(interval)* returns the set of all nodes in an interval; the interval itself is a tuple.

Given the sets *varsthru* and *defsfrom* for basic blocks, we compute corresponding sets for all the intervals of the derived sequence. Three sets are used in an auxiliary way.

1. *varsreaching(b)* - the set of variables for which there exists a defintion-clear path from the interval entry to b.
2. *defsreaching(b)* - initially, the set of definitions within an interval which can reach b by a path not containing a backwards branch; later, the set of all definitions which can reach b.
3. *defhead* - the set of definitions within the interval which can reach the head via a latch.

A SETL algorithm for the first phase of the use-definition chaining process is presented below. Note that since nodes within an interval are processed in interval order, each of the assignment statements occurring in our algorithm has a well defined outcome, in that all necessary sets are appropriately extended before they are used; also note the algorithm admits an efficient bit-vector realization (using "and" for intersection (*), and "or" for union (+)).

```
/* first a group of three auxiliary macros */
macro exits(int,sint);
   = {pb ∈ pred(sint(1))|pb ∈ nodesof(int)}; endm exits;
macro defsthru(pb,b);
   = {d∈defsreaching(pb)|var(d)∈varsthru(pb,b)}; endm defsthru;
macro latches(intv);
   = {n∈nodesof(intv)|head ∈ cesor(n)}; endm latches;
/* the argument to the routine which follows is a collection of
   intervals constituting one of the graphs in the derived sequence*/
define usedefpass1(intervals);
/* cesor, pred, varsthru, defsfrom, var, defs, vars, nodesof,
   varsreaching, and defsreaching are all assumed to be global */
(∀intv ∈ intervals);
   head = intv(1);
   defsreaching(head) = nl; varsreaching(head) = vars;
   (2 ≤ ∀i ≤ #intv)  /* process interval */
      b = intv(i);
      varsreaching(b) = [+: pb ∈ pred(b)]
```

```
            (varsreaching(pb) * varsthru(pb,b));
      defsreaching(b) = [+: pb ∈ pred(b)]
            (defsfrom(pb,b) + defsthru(pb,b));
   end ∀i;
   defhead = [+: ℓa ∈ latches(intv)]
            (defsfrom(ℓa,head) + defsthru(ℓa,head)) orm nℓ;
   defsreaching(head) = defhead;
   (∀sint ∈ cesor(intv))
      sb = sint(1) /* head of successor interval */ ;
      varsthru(intv,sint) = [+: ex ∈ exits(intv,sint)]
            (varsreaching(ex) * varsthru(ex,sb)) orm nℓ;
      defsfrom(intv,sint)  =
            {d∈defhead|var(d)∈varsthru(intv,sint)}
                  + [+:pb∈ exits(intv,sint)]
                        (defsfrom(pb,sb) + defsthru(pb,sb))orm nℓ;
   end ∀sint;
end ∀intv;
return;
end usedefpassl;
```

The above routine will calculate *varsthru* and *defsfrom* for the intervals of a control flow graph and all its derived graphs.

The second pass of the overall use-definition chaining process recalculates the set *defsreaching(b)* for each block b, causing this set to include all definitions D which can reach b by any path (free of redefinitions of the variable defined by D) in the program graph. This second pass is shown as part of the following SETL algorithm, which also represents the overall use-definition chaining process:

```
/* the argument to this routine is the derivation sequence of
   an originally given graph */
define usedf(seqd);
   /* cesor, pred, varsthru, defsfrom, var, defs, vars, nodesof,
   varsreaching, defsreaching are assumed to be global */
varsreaching = nℓ; defsreaching = nℓ; /* auxiliary functions
                                 are initially undefined */
   < ∀n ≤ #seqd) intervals = hd seqd(n); /* call passl */
   usedefpassl(intervals);;
/* assign defsreaching for final derived graph */
```

```
defsreaching (∋hd seqd(#seqd)) = initial;
/* proceed through  second pass, in reverse order of derivation
   sequence */
(#seqd > ∀n > 1)
   intervals = hd seqd(n);
   (∀intv ∈ intervals)
      defsreaching(intv(1))=defsreaching(intv)+defsreaching(intv(1));
      /* process nodes in interval order */
      (2 ≤ ∀i ≤ #intv)
         b = intv(i);
         defsreaching(b) =
            [+: pb∈pred(b)](defsfrom(pb,b)+defsthru(pb,b));
      end ∀i;
   end ∀intv;
end ∀n;
return;
end usedef;
```

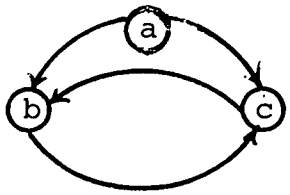
Note that as it stands the algorithms which have just been given are valid only for program graphs which are eventually reducible. It is also worth observing that the set

uses(entry)

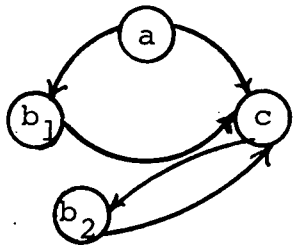
i.e., the set of variables which are live immediately on entry to the program, is in fact the set of improperly initialized program variables, concerning which we would, in a total optimizing-compiler system, wish to issue a diagnostic message.

Node splitting; An algorithm for live-dead analysis including node splitting.*

If no irreducible graphs (consisting of more than one point) are encountered, the derivation sequence of an originally given program graph g can be used in the manner illustrated by the last few algorithms. Irreducible graphs are obstacles to the application of these algorithms as they stand. Fortunately, easy generalizations will handle irreducible graphs. To see what generalizations are necessary, note that the simplest example of an irreducible graph is as follows:



If for the moment we think of the code represented by the node b as being duplicated into two copies, one of which, b_1 , is entered from a, while the other, b_2 , is entered from c, then we see that there exists a program, equivalent to that having the flow depicted above, but having the following flow instead.



In this graph $\{a,b_1\}$ and $\{c,b_2\}$ are intervals; so the derived graph of this graph consists only of two nodes, and its second derived graph consists only of one. This example shows that an abstract process of "node splitting" will allow the reduction even of graphs which originally are irreducible. Moreover, the phenomenon observed in this example is perfectly general. To see that this is the case,

*Some of the material in the present subsection is adapted from an algorithm given in SETL Newsletter 38 by K. Kennedy.

we argue as follows. Let g be a program graph consisting of n nodes. Let s be a set of nodes within g, including the entry node of g, consisting of as few nodes as possible, and such that when s is removed, the remaining set $\bar{s}$ of nodes admits no loops; which is to say that at least one point of every loop belongs to s. Note that since every loop contains at least 2 nodes, s will never have more than n-1 points. For each $x \in s$, let f(x) be the set of all points in $\bar{s}$ reachable along a path starting at x, all the nodes of the path but x lying in $\bar{s}$. We then construct a *split graph*, as follows. The points of this new graph are the pairs $\langle x, n\ell \rangle$ for $x \in s$, together with the points $\langle y,x \rangle$ where $y \in f(x)$; these points represent multiple "split copies" of y. In this collection, we define the successor relationship as follows:

i. cesor($\langle x, n\ell \rangle$) is the set of points $\langle y,x \rangle$ for which $y \in$ cesor(x), together with the set of points $\langle y, n\ell \rangle$ for which $y \in$ cesor(x);

ii. cesor($\langle y,x \rangle$) is the set of points $\langle z,x \rangle$ for which $z \in$ cesor(y), together with the set of points $\langle z, n\ell \rangle$ for which $z \in$ cesor(y).

It is clear from this definition that, in the split graph, each of the points $\langle y,x \rangle$, $y \in f(x)$, can be reached from $\langle x, n\ell \rangle$; moreover, the set $\{\langle y,x \rangle, y \in f(x)\}$ contains no cycles, since, if it did, so would $\bar{s}$, which by construction we have ruled out. Therefore each of the sets $\{\langle y,x \rangle, y \in f(x)\}$ with x is an interval. Hence our split graph can be covered by at most n-1 intervals, and its derived graph will therefore contain at most n-1 points. Thus, by applying the node splitting process whenever an irreducible graph is encountered, we can construct a generalized derivation sequence which will always converge to a graph consisting of only one single node.

To find, within a graph g, a set s of nodes of the kind required, we may proceed as follows.* Arrange the nodes of g in sequence, using the ordering algorithm described in the first part of the present section. Then define s as follows: first put all the

* More efficient node-splitting algorithms have been designed by J. Cocke and R. Miller.

predominators of any node of g into s. Then take all backward branches whose origin node u does not belong to s; suppose that the target node of the branch is v. If every predominator of u is a predominator of v, put v in s.

In SETL, this algorithm and the remaining parts of the construction of the split graph appear as follows. (Note the use in what follows of the transitive closure function *closure*, defined on page 119).

```
definef graphsplit(nodes,entry); /* cesor is assumed to be global*/
seq = graphord(nodes,entry);
domsof = predoms(nodes,entry); s = [+: x∈domsof]x(2) orm nl;
seqno ={< seq(j),j>, 1 ≤ j ≤ #seq};
(#seq ≥ ∀j ≥ 2 | n seq(j) ∈ s)
   u = seq(j);
   (∀v ∈ cesor(u) | seqno(v) lt j)
      if domsof{v} incs domsof{u} then b in s;;
   end ∀v;
end ∀j;
sbar =  nodes - s;
newg = {<x,nl>, x∈s} + {<y,x>, x∈s, y∈closure(cesor,{x},sbar)};
(∀x ∈ s)
   cesor(<x,nl>) = {<y, if y sbar then x else nl>, y∈cesor(x)};
   (∀y ∈ closure(cesor,{x},sbar))
      cesor(<y,x>) = {<z, if z∈sbar then x else nl>,z ∈cesor(y)};
   end ∀y;
end ∀x;
newent = <entry, nl>;
return newg;
end graphsplit;
```

In regard to the use of a split graph for the type of analysis in which we will be interested, as for example in the derivation of live-dead information, we may make the following remarks. In the generalized version of the routine *builda*, each of the nodes <y,x> derived from y by splitting will inherit the set *uses* of variables used within it from y, i.e., we will have uses(<y,x>) = uses(y). Similarly, we will have thru(<y,x>,<$\bar{y}$,x>) = thru(y,$\bar{y}$) if y has the successor $\bar{y}$ in $\bar{s}$, thru(<y,x>,<$\bar{y}$,nl>) = thru(y,$\bar{y}$) if y has the successor $\bar{y}$ in s, etc. Then in the routine *builduse*,

we will obtain use(y) for a node that must be split into nodes $y_1,\ldots,y_n$ as the union of $use(y_1),\ldots,use(y_n)$.

We shall now describe a procedure for live-dead analysis alternate to that presented previously, and go on to indicate the generalizations necessary to handle irreducible graphs.

A path in the control flow graph of a program is said to be *definition-clear* (or def-clear) with respect to a variable A if there is no definition of A between the beginning and end of the path.

The algorithm to be described will provide live-dead information for each block in a program. This information will be provided by giving the set *live(block)* of variables which are live on entry to a block. The initially given information used by our algorithm will be as follows:

1. For each block, we will be given the set of all variables which are live on entry to that block in virtue of the existence of a definition-clear path from the block entry to a use within the block itself. We shall call this set *blkuses(block)*.

2. For each block, we will be given the sets *thru(block,sblock)*, one for each *sblock* $\in$ *cesor(block)*; *thru(block,sblock)* is the set of variables for which there is a definition-clear path, from the entry to *block*, through *block*, to *sblock*. (In the discussion which follows, *cesor(b)* denotes the set of all basic blocks which are immediate successors of block b, and *pred(b)* denotes the set of all immediate predecessors of b.)

There is an important relation among these three sets (*live, thru, blkuses*), which can be explained as follows: a variable is live on entry to a block if it is live by virtue of a definition-clear path to a use within the block or if there is a definition-clear path through the block to a successor block at whose entry the variable is live. This relation can be expressed by the following set relation:

$$(1) \qquad live(block) = blkuses(block) \cup \left(\bigsqcup_{sb \in cesor(block)} (thru(block,sb) \cap live(sb) \right)$$

-- as a SETL "code fragment":

(1') live(block) = blkuses(block) + [+: sb ∈ cesor(block)]
(thru(block,sb) * live(sb));

It is easy to see that, using this relation, we will be able to find out which variables are live on entry to a block if we know which variables are live on entry to its successors. This idea is the basis for our algorithm.

The algorithm will proceed in two passes over the nodes of the control flow graph and all its derived graphs.

1. The first pass will compute the *thru* and *blkuses* sets for intervals of the derived graphs.

2. The second pass will compute the *live* set, first for the single node of the last derived graph, then for each node in the underlying interval. It will continue in this manner until the *live* sets have been computed for each node in the control flow graph.

The "thru" and "blkuses" information for a basic block b can be derived by examining b. The essential work of the first pass lies in computing these sets for an interval, given the sets for each node in that interval. This can be done as follows. Suppose we have two auxiliary sets, *path(block)* and *insidesofar*. The set *path(block)* contains all variables for which there is a definition-clear path from the interval entry to the entry of *block*. The set *insidesofar* is an accumulator set. When, in processing the blocks of the interval in interval order, the block b is processed, we add to *insidesofar* all variables for which there is a def-clear path from interval entry to b and which are in the set blkuses(b). These are the variables which will be in the set *blkuses(interval)*. Hence to process b we execute

(2) insidesofar = insidesofar ∪ (path(b) ∩ blkuses(b))

This equation is used for all blocks in the interval.

There is a def-clear path for a variable from interval entry to a block b if there is such a path from interval entry to some predecessor of b and if there is a def-clear path through the predecessor to b. Hence

(3) $$path(b) = \bigsqcup_{pb \in pred(b)} (path(pb) \cap thru(pb,b))$$

Note however that this relation does not hold for the head of the interval; since entry to the head is identical with entry to the interval, we have instead:

(4) $$path(head) = \text{all variables}$$

Suppose J is an interval which is a successor of I and that j_1 is its head. The node j_1 must be a successor of at least one block in I. We can therefore compute thru(I,J) as follows:

(5) $$thru(I,J) = \bigsqcup_{b \in pred(j_1) \cap I} (path(b) \cap thru(b,j_1))$$

As previously indicated,

(6) $$blkuses(I) = insidesofar$$

where the right-hand side denotes the value of *insidesofar* after all nodes in the interval have been processed.

The algorithm which follows represents the 'innermost' part of an overall live-dead analysis procedure. Given the values of *thru(a,b)*, it associates sets *blkuses* and *path* with all the nodes of an interval. It will process the nodes of the interval I in interval order, computing the sets *path* and *insidesofar*. Because it uses interval order, the predecessors of a node are always processed before the node itself and the path sets required by equations (2) and (3) are available when needed. Note that we need not worry about the contribution of loops within the interval, because a loop cannot contribute any new paths. This is because *path(head)* is already as large as it can be.

The final step of our algorithm will be to compute the *thru* and *blkuses* sets for the interval, using equations (5) and (6).

The following SETL algorithm, *process(interval)*, performs the computations specified above. Its argument, *interval*, is assumed to be a SETL tuple of nodes, with interval(1) equal to the interval head.

```
define process(interval);
/* pred, cesor,blkuses, thru, and allvars are assumed to be global*/
path = nl,  insidesofar =  blkuses(interval(1));
path(interval(1)) = allvars;
/* pass through the interval in interval order */
```

```
(? ≤ ∀i ≤ #interval)
   b = interval(i);
   path(b) = [+: pb∈pred(b)](path(pb)*thru(pb,b)) orm nℓ;
   insidesofar = insidesofar + path(b) * blkuses(b);
end ∀i; /* now calculate thru and blkuses sets for the interval */
blkuses(interval)= insidesofar;
(∀y ∈ cesor(interval))
   preint = pred(y(1))*{nodes,nodes(j)∈interval};
   thru(interval,y) = [+: b ∈ preint](path(b)*thru(b,y(1))) orm nℓ;
end ∀y;
return;
end process;
```

The function orm used above is, as before, defined by

```
definef a orm b; return if a ne Ω then a else b; end orm;
```

Having described this basic process, we may now go on to explain the overall structure of the first pass of our live-dead analysis. In this pass, we process the elements of the derived sequence of a program graph, starting with basic intervals, then intervals of the first derived graph, and so on until a final interval, which is the fully reduced control flow graph. We merely call the routine *process* for each interval in all derived graphs, as is shown in the following code:

```
/* seqd is the derivation sequence of an initially given program
   graph */
define pass1(seqd);
(1 < ∀i < #seqd)
process[hd seqd(i)];;
return;
end pass1;
```

At the end of pass1, we will have computed the *blkuses* and *thru* sets for each interval in the sequence of derived graphs, including final interval which reduces the program to a single node. Since an exit block has no successors, the *live* set for an exit block is

equal to the *blkuses* set for that block. Since the single node occurring last in the derived sequence and representing the whole program is an exit block, its *blkuses* set is the set of variables which are live on entry to the program. This set is useful in two ways. First, it tells us which variables are improperly initialized in the program; second, it is used in computing the live sets for nodes of the underlying interval.

The overall structure of our dead variable trace is shown by the following code, in which *seqd* represents the derivation sequence of an initially specified program graph:

```
define livevars(seqd);  /* live and blkuses are global */
pass1(seqd);  /* call preceding pass1 process */
/* establish live set for the single node  to which the
   derivation sequence ultimately reduces */
live(∋hd seqd(#seqd)) = blkuses(∋hd seqd(#seqd));
pass2(seqd);
return;
end livevars;
```

This 'driving routine' uses the *pass1* procedure which we have just depicted to calculate the set of variables live upon program entry, and then calls pass2. The subroutine pass2, which we now show, simply applies the routine *liveint* (described below) to each interval in the derived graph sequence, starting with the last derived graph and working forward. This order will insure that we always process outer intervals before we process inner intervals.

```
define pass2(seqd);
(#seqd ≥ ∀i > 1) liveint[hd seqd(i)];
return;
end pass2;
```

The routine *liveint* merely calculates the *live* sets for every node in an interval, given the live sets for the entry to the interval and for the entries to all successors of the interval. It does s

by processing the nodes of the interval in reverse interval order
and calculating the *live* set for each node encountered. The formula used is (1). In order to use this formula, we must have *live* sets for each successor of the node we are processing. But in virtue of the processing order used we do have these sets. Indeed, suppose that we are examining a node b and a particular successor sb. There are three possibilities.

1. If sb is not in the interval I containing b, it must be the head of some successor interval y. Since we have computed the *live* sets for every successor interval and since entry to an interval is the same as entry to its head, we can use *live(y)* for *live(sb)*.

2. If sb is in I but is not the head, we must have already processed sb and so will have computed *live(sb)* already.

3. If sb is the head of I, we can use *live(interval)* since entry to I is the same as entry to sb. Recall that *live(interval)* will always have been determined before we process the nodes of that interval.

Thus we are always in a position to apply formula (1).

The routine *liveint* in SETL is as follows:

```
define liveint(interval);
/* cesor, pred, thru,blkuses, live are all assumed to be global */
/* pass through the interval in reverse order */
live(interval(1)) = live(interval);
(#interval ≥ ∀i ≥ 2)
   b = interval(i);  live(b)= blkuses(b);
   (∀sb ∈ cesor(b))
     live(b)=live(b) + (thru(b,sb) *
       (if ∃y∈cesor(interval)|sb eq y(1) then live(y)else live(sb));
   end ∀sb;
end ∀i;
return;
end liveint;
```

On completion of pass2, this subroutine will have computed a *live* set for each block in the control flow graph.

We shall now incorporate nodesplitting into the dead variable analysis procedure that has just been presented. The nodesplitting technique used here is that sketched in the first paragraphs of the present subsection. If one of the derived graphs G_j of a program graph G cannot be reduced further, it is transformed to give a graph $\bar{G}_j$ in which several of the nodes of G_j are split into more than one copy. In this case, $\bar{G}_j$ replaces that of G_j in the derivation sequence. The nodes of the graph $\bar{G}_j$ are SETL pairs, where the first item of each pair is a node of the original graph G_j and the second item of each pair is either $\underline{n\ell}$, in the case of an unsplit node, or another node of the graph G_j , in the case of a split node. The successor function for the transformed graph G_j is as described earlier in the present subsection. To allow detection of split graphs, each index j for which G_j has been replaced in the derivation sequence by $\bar{G}_j$ is assumed to have been made a member of a global set *splitgraphs*.

To modify the dead variable analysis algorithm appropriately we need to know:

1. How to derive *thru* and *blkuses* sets for the nodes of $\bar{G}_j$ given these sets for the nodes of G_j (this information is needed in the first pass), and

2. How to derive the *live* sets for the nodes of G_j given these sets for the nodes of $\bar{G}_j$ (this information is needed in the second pass).

To answer these questions, consider the manner in which the nodes of $\bar{G}_j$ are defined. Let b and sb be nodes in $\bar{G}_j$ such that sb ∈ cesor(b). The node b represents a (possibly split) 'copy' of the code in the block $\underline{hd}$ b; similarly, sb represents a 'copy' of the code in the block $\underline{hd}$ sb. If there is a definition-clear path, from the entry of $\underline{hd}$ b, to a use of a variable in $\underline{hd}$ b, there must be a definition-clear path from the entry of b to a use in b of the same variable. Therefore,

(1) blkuses(b) = blkuses($\underline{hd}$ b);

This defines *blkuses(b)*. For the same reason, if b and sb are nodes of $\bar{G}_j$, we put

(2) thru(b,sb) = thru($\underline{hd}$ b, $\underline{hd}$ sb);

These formulas are used if nodesplitting must be employed. The responding formula to be used during the second pass is slightly more complicated. If $b_1, b_2, \ldots, b_n$ are nodes in $\bar{G}_j$ which all represent copies of the code within hd b_1 , then if a variable is live on entry to any of the nodes $b_1, b_2, \ldots, b_n$ it must be live on entry to the node hd b_1 in G_j. Therefore we have

(3) live(hd b_1) = [+: $1 \leq i \leq n$] live(b_i)

This transformation will be used in the generalized routine *liveint* shown below: in calculating live(b) a flag *isplit* will tell us when we deal with a split graph; we will always execute the following SETL statement:

(4) if isplit then live(hd b) = live(b)+live(hd b) orm nl; .

This will ensure that the summation shown in (3) is performed, and that the correct value will have been assigned to live(hd b) for each node b of $\bar{G}_j$ by the time processing of $\bar{G}_j$ is terminated. The transformations (1) and (2) will be applied in an initialization block at the beginning of the routine *process*.

The modifications described above suffice to specify a correct live-dead analysis procedure for a program whose derivation sequence includes split graphs.

The following SETL algorithm incorporates these modifications. The reader will note that it is identical to the algorithm presented earlier except for the insertion of the transformations described above.

```
define process(interval);
/*  pred, cesor,blkuses, thru, allvars, isplit are global */
path = nl;  insidesofar = nl;
path(interval(1)) = allvars;
/* test for split  cases*/
if isplit then
   (1 ≤ ∀i ≤ #interval)
        b = interval(i); blkuses(b)=blkuses(hd b);
        (∀sb ∈ cesor(b)) thru(b,sb) = thru(hd b, hd sb);;
   end ∀i;
end if;
```

```
insidesofar =blkuses(interval(1));
/* pass through the interval in interval order */
(2 ≤ ∀i ≤ #interval)
   b = interval(i);
   path(b) = [+: pb∈pred(b)](path(pb)*thru(pb,b));
   insidesofar = insidesofar + (path(b) *  blkuses(b));
end ∀i;
/* now calculate thru and inside for the interval */
blkuses(interval)= insidesofar;
(∀y∈cesor(interval))
   preint=pred(y(1)) * {n|n(j)∈interval};
   thru(interval,y) = [+: b∈preint](path(b) * thru(b,y(1))) orm nℓ;
end ∀y;
return;
end process;

define pass1(seqd); /*isplit and splitgraphs are assumed global*/
(1 < ∀i < #seqd)
   isplit = (i-1) ∈ splitgraphs;
   process[hd seqd(i)];
end ∀i;
return;
end pass1;

define livevars(seqd)
/* live, blkuses   are assumed to be global */
pass1(seqd);
/* set live set for the single node */
live (∋hd seqd(#seqd)) = blkuses(∋hd seqd(#seqd));
pass2(seqd);
return;
end livevars;

define pass2(seqd);/*isplit and splitgraphs are assumed global*/
(#seqd ≥ ∀i > 1)
isplit = (i-1) ∈ splitgraphs; liveint[hd seqd(i)];
end ∀i;
return;
end pass2;
```

```
define liveint(interval);
   cesor, pred, thru, blkuses, live and isplit are global */
/* pass through the interval in reverse order */
live(interval(1)) = live(interval);
(#interval ≥ ∀i ≥ 2)
   b = interval(i);  live(b) = blkuses(b);
   (∀sb ∈ cesor(b))
    live(b) = live(b) + (thru(b,sb)) *
     (if ∃y∈cesor(interval)|sb eq y(1) then live(y) else live(sb));
   end ∀sb;
/* test for split nodes */
   if isplit then
      live(hd b)=live(b) orm nℓ+(live(hd b) orm nℓ);
   end if;
end ∀i;
return;
end liveint;
```

The above represents the complete live-dead analysis algorithm with node-splitting included, in a form which is suitable for implementation as part of an optimizing compiler based on interval techniques.

5. An Algorithm for Redundant Expression Elimination and Code Motion.*

We will now describe a redundant expression elimination algori . This algorithm will include both 'profitability of motion' and 'safety of motion' considerations. The approach used will be a new one: we shall handle all code motion and 'hoisting' by inserting computations at interval entries, allowing the common expression elimination routine to do the rest.

'Safety' (see below) will be handled by a systematic analysis similar to that used in the live-dead trace algorithms described in the preceding pages. Profitability is handled using a somewhat heuristic scheme based on the assumption that it is profitable to move code out of strongly-connected regions.

To make the structure of our algorithm plain, we will now describe various of its aspects separately: first a simple calculation-redundancy algorithm will be described; then we will show how this algorithm can be generalized to handle code motion; and a final generalization will yield an algorithm incorporating safety considerations as well.

i. Redundant Expression Elimination (availability computation)

To accomplish redundant expression elimination, we compute sets *avail(b)* for each block b in the control flow graph. For a given block b, *avail(b)* is the set of expressions which are always available on entry to b. Note that we say that an expression is *available* at a given point in a program if, along every path leading to this point, there will be found a computation of the expression not followed by any reassignment of the value of one of its inputs. This implies that the computation is redundant, since a previously computed value of the same expression is immediately available for reloading. Once we have the *avail* sets we can determine those expressions which are redundant within a basic block by a simple linear scan.

How can we compute the *avail* sets? Suppose that an initial scan of the basic blocks produces the following sets associated

---

*Most of the material in this subsection is taken from SETL Newsletter 28 by K. Kennedy.

with a block b and its immediate successors sb.

expdown(b,sb) the set of expressions which are "downward exposed" on a path through b to sb; that is, the set of expressions which are computed on each path through b to sb and are not subsequently "killed" by a redefinition of one of their arguments. Note that *expdown(b,sb)* contains those subexpressions whose values are always available on exit from b to sb, irrespective of the code preceding b.

2. *nokill(b,sb)* the set of expressions which are available on exit to sb from b whenever they are available on entry to b. Note that, since every element of *expdown(b,sb)* is unconditionally available on exit to sb from b we have *expdown(b,sb)* $\subset$ *nokill(b,sb)*.

Using these sets, we are able to state a formula for the *avail* sets. If a subexpression is to be available on entry to block b, it must be available on exit from each of its predecessors pb.
To be available on exit from pb, it must either be downwards exposed in pb or it must be available on entry to pb and not killed in pb. These conditions are expressed by the following SETL code fragment.

```
(1)  avail(b) = [*: pb ∈ pred(b)](nokill(pb,b)*avail(pb)+expdown(pb,b));
```

This formula expresses the *avail* set for b in terms of the *avail* sets for the predecessors of b. An expression of this kind implies that analysis should start at the beginning of the program and proceed forward through the basic blocks. However, the term "forward" means little in a graph with cycles unless we take advantage of the Cocke-Allen interval partition. We know that we can assign a total order to the nodes of an interval; therefore we compute availability within an interval conditionally on assumptions which must be made concerning availability on entry to the interval.

Suppose that *comps* is the set of all expressions under consideration. One of two extreme assumptions might be made about *avail(interval)*:

i. all subexpressions are available on entry to the interval.
ii. no subexpressions are available on entry to the interval.

Corresponding to these two assumptions, we define two new sets:

1. *posavail(b)* the set of expressions which are available on entry to b, assuming all expressions are available on entry to the interval containing b.
2. *defavail(b)* the set of expressions which are available on entry to b, assuming that no expression is available on entry to the interval containing b.

Then *avail(b)* can be expressed as a linear combination of these sets. An expression is available on entry to b if it is in *defavail(b)* or if it is in *posavail(b)* and is available on entry to the interval containing b.

(2) avail(b) = avail(interval) * posavail(b) + defavail(b);

The auxiliary sets, *posavail* and *defavail*, are easy to compute if we process the nodes of the interval in interval order, applying the appropriate analogs of equation (1) at each node. Interval order is important because equation (1) requires that we compute the *posavail* and *defavail* sets for all predecessors of a node before processing that node. If we assume that each *interval* is the sequence of its nodes in interval order (i.e., *interval*(1) is the head of *interval*, etc.), then computation can proceed as follows.

```
/* initialization first */
posavail(interval(1)) = comps;
defavail(interval(1)) = nℓ;
/* pass through nodes in interval order */
(2 ≤ ∀i ≤ #interval)
   b = interval(i);
   posavail(b) = [*: pb ∈ pred(b)]
         (posavail(pb) * nokill(pb,b) + expdown(pb,b));
   defavail( b) = [*: pb ∈ pred(b)]
         (defavail(pb) * nokill(pb,b) + expdown(pb,b));
end ∀i;
```

This code is not enough, however, because it does not take account the possibility that, because of branches to the head from within the interval, a computation available on entry to the interval may not be available on entry to its head. Let *headav* be the set of all expressions which will be available on entry to the head of the interval, on the assumption that all computations are available on entry to the interval.

```
(3)  headav=[*: pb ∈ (pred(interval(1)) * nodes(interval))]
                  (posavail(pb) * nokill(pb,interval(1))
                              +  expdown(pb,interval(1)));
```

In a full computation of computation availability, we must modify each *posavail* set to reflect the fact that only the computations in *headav* can actually be available on interval entry. This can be done by applying the analog of equation (2) above, i.e., by using the following SETL code sequence.

```
/* modify posavail sets */
(2 ≤ ∀i ≤ #interval)
   b = interval(i);
   posavail(b) =headav * posavail(b) + defavail(b);
end ∀i;
```

(Note that interval order is not crucial to this last operation.)

Finally, in order to be able to iterate the above-described process for higher order intervals, we must compute *expdown* and *nokill* sets for each interval. Suppose *sint* is an immediate successor of *interval*. An expression is in *expdown(interval,sint)* if, for every predecessor b of *sint(1)* in *interval*, either the expression is downwards exposed in b or the expression is in *defavail(b)* and is not killed in b. Hence the following code fragment:

```
/* compute expdown for interval */
expdown(interval,sint) = [*: b∈(pred(sint(1)) * nodes(interval))]
      (defavail(b) * nokill(b,sint(1)) + expdown(b,sint(1)));
```

Similarly, an expression is in *nokill(interval,sint)* if, for ry predecessor b of *sint(1)* in *interval*, it is either in *posavail(b)* and not killed in b or it is downwards exposed in b .

```
/* compute nokill for interval */
nokill(interval,sint) = [*: b ∈ (pred(sint(1)) * nodes(interval))
      (posavail(b) * nokill(b,sint(1)) + expdown(b,sint(1)));
```

The code fragments which have been shown will now be incorporated into a SETL routine *redproc1(interval)*. To abbreviate this routine, which is shown below, we use a function *transf* which simply applies the transformation shown by equation (1).

```
definef transf(pb,b,tset);
/* expdown, nokill are global */
/* compute linear transition */
return (tset(pb) * nokill(pb,b) +expdown(pb,b));
end transf;
```

The routine *redproc1* accepts, as its only argument, a sequence of nodes (in interval order) constituting an interval.

```
define redproc1(interval);
/* cesor, pred, defavail, posavail, comps, nokill, expdown
   are all assumed to be global */
/* initialize for processing */
head = interval(1);
defavail(head) = nl;
posavail(head)=  comps;
/* define the set of nodes in the interval */
intnodes = {interval(i), 1 ≤ i ≤ #interval};
/* the first pass to get defavail and an initial estimate of
   of posavail */
(∀b(i) ∈ interval | i ge 2)
   preds = pred(b);
   defavail(b) = [*: pb ∈ preds] transf(pb,b,defavail);
   posavail(b) = [*: pb ∈ preds] transf(pb,b,posavail);
end ∀b(i);
/* compute the set of subexpressions actually available on entry
   to the head */
headav= [*: pb ∈ (pred(head) * intnodes)] transf(pb,head,posavai
posavail(head) = headav;
```

```
/* recompute the posavail sets */
   ›(i) ∈ interval | i ge 2)
   posavail(b) =headav * posavail(b) + defavail(b);
end ∀b(i);

/* compute expdown and nokill sets for the interval */
(∀sint ∈ cesor(interval))
   preds = pred(sint(1)) * intnodes;
   expdown(interval,sint) = [*: b ∈ preds]transf(b,sint(1),defavail);
      nokill(interval,sint)=[*:b∈preds]transf(b,sint(1),posavail);
end ∀sint;
end redprocl;
```

To calculate expression availability (i.e. redundancy) we apply the preceding routine, first to each interval in a control flow graph, then to each interval in the first derived graph, and so on until we reach a highest order graph which we assume reduces the program to a single interval I. Since no expression is available on entry to the program, we have

avail(I) = nℓ;

as an initial equation. We use this fact and equation (2) to compute *avail* sets for the nodes of the highest order then for the next highest order and so on, making sure that the *avail* set is computed for each interval entry before the nodes of that interval are processed. This process will terminate when *avail(b)* has been computed for every block b in the program.

The following algorithm, embodying the above observation, computes *avail* sets for the nodes of an interval, given the *avail* set for the interval. Its only input argument, *interval*, is a sequence of nodes (given in interval order).

```
define optcess(interval);
/* defavail, posavail, and avail are assumed to be global */
/* apply equation (2) */
(∀b(i) ∈ interval)
   avail(b) = avail(interval) * posavail(b) + defavail(b);
    ∀i;
end optcess;
```

To 'drive' the routines described above we use a routine *csx* (for common subexpression elimination) which accepts an input argument *seqd*; this argument is the sequence of all derived graphs of a control flow graph. Thus *seqd(#seqd)* is the highest order derived graph, which we assume reduces the whole initially given flow to a single node. The routine *csx*, which follows, merely applies the routines *process* and *optcess* in an appropriate order.

```
define  csx(seqd);
/* avail is assumed to be global */
/* inner-to-outer pass to compute posavail and defavail */
(∀graph(k)∈seqd|k ge 2)(∀int∈hd graph)redprocl(int);;;
/* nothing available on entry to program */
avail(∋hd seqd(#seqd))= nl;
/* outer-to-inner pass to compute availability */
(#seqd ≥ ∀k ≥ 1) (∀int∈hd seqd(k)) optcess(int);;;
return;
end csx;
```

This completes our description of an initial, simplified, common subexpression elimination algorithm. The algorithm which we have just described will now be used as a framework into which an additional process of code motion will be hung.

ii. Code Motion (with "Profitability")

Motion of code out of loops will be accomplished by redundantly inserting, at interval entries, computations discovered to be movable. The 'redundancy' or 'availability' algorithm just described then allows us to remove the same computations from blocks within intervals. 'Code motion' will result from the combination of these two steps, i.e. will result from insertion followed by deletion.

An expression e is *movable* out of a block b if it is *upwards exposed* on some path P through the block; that is, if e appears on P and is not preceded along P by a redefinition of any of its arguments. If in this situation we wish to refer to P, we will say that e is upwards exposed *on P*. Let sb be a successor

block of b. Then *movable(b,sb)* denotes the set of expressions e ich are upwards-exposed in b on some path leading to sb.

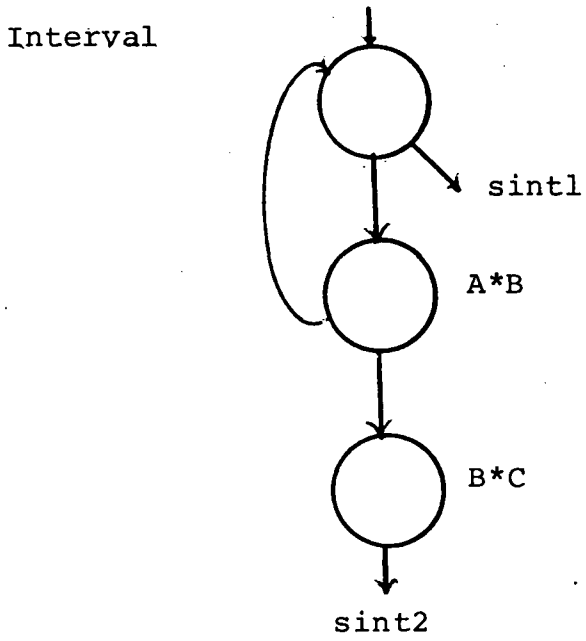
The second argument sb introduced here is used to take account of 'profitability' considerations. We make the assumption that, when motion is possible, it is profitable to move expressions out of the strongly-connected subpart of an interval, but not out of the remainder of the interval. The argument sb of *movable(b,sb)* helps us determine if an expression is in the strongly-connected region SCR associated with an interval; indeed, an expression in *movable(b,sb)* lies in SCR if sb is in SCR. We need to make use of a similar argument in all levels of processing since it might well be desirable to move, out of a high-order interval, an expression which it is not profitable to move out of the interval which directly contains the expression.

Our code motion algorithm is structured into two passes. We compute the *movable* sets for each of the intervals of a sequence of derived graphs during a first, inner-to-outer pass. On the second, outer-to-inner, pass we compute sets *insert(interval)* containing the expressions to be computed on entry to *interval*. This second computation uses a heuristic: *insert(interval)* includes those expressions which are movable out of the interval and which appear along paths to every successor of the interval. Note that every movable expression in the strongly connected subpart of an interval satisfies this criterion: there is certainly a path from such a calculation to every successor interval (since from a node in the strongly connected part of an interval there is always a path to the interval head).

In SETL notation, the set of expressions which our heuristic selects appears as follows:

```
[*: sint ∈ cesor(interval)] movable(interval,sint)
```

As an example of the working of our heuristic consider the following interval I:



Suppose that A*B and B*C are both movable. Then A*B is in both *movable(interval,sint1)* and *movable(interval,sint2)*. Therefore a calculation of A*B will be inserted at the entry to I. However, B*C belongs only to *movable(interval,sint2)* so it will not be inserted at the entry to I. However, it may be inserted at the entry to a higher order interval containing I.

Note finally that we will not wish to insert any expression at a point at which it is already available. This consideration leads us to the following formula for the set *insert(interval)*.

(4) insert(interval) = [*: sint∈cesor(interval)]movable(interval,sint)
- avail(interval);

We observe at this point that, in the algorithm which follows, program exits are represented by special pseudo-blocks which can never be incorporated into any interval. Consequently every interval appearing in the algorithm has at least one successor.

By changing the routine *optcess* encountered above the calculation represented by equation (4) can be incorporated into the second pass of the skeletal availability calculation described in the preceding pages. In the algorithm which results, we compute *insert(interval)* using equation (4) before processing the individual nodes; the expressions belonging to this set are adde to those in *avail(interval)* to form a set *newavail*. The set *new* il

contains all expressions which are available on entry to the interval after the expressions in *insert(interval)* have been placed in their new position. The new *optcess* routine is as follows:

```
define  optcess(interval);
/* avail, posavail, defavail, movable, insert and cesor are global*/
/* compute insert set using equation (4) */
insert(interval) = ([*:sint∈cesor(interval)]movable(interval,sint))
                                                    - avail(interval);
/* add insert set to availability set for interval entry */
newavail = avail(interval) + insert(interval);
/* compute the avail sets for nodes of the interval */
(∀b(i) ∈ interval)
   avail(b) = posavail(b) * newavail + defavail(b);
end ∀b(i);
end optcess;
```

We now turn to describe the manner in which *movable* sets will be calculated for intervals by an expanded version *redproc2* of the *redproc1* subroutine. Expressions which are movable out of b and are in *posavail(b)* but not in *defavail(b)* are candidates for code motion. Indeed, such expressions are redundant at b if and only if they are available on interval entry; by inserting them at interval entry we make them redundant at b. However, profitability enters as a complicating consideration. Let *sint* be an immediate successor of *interval*. We include an expression C in *movable(interval,sint)* if there exists a block b of *interval* such that C is in *posavail(b) - defavail(b)*, and such that C is also in *movable(b,sb)* where sb is either the head of *sint* or a block from which *sint* can be reached by a path within *interval*. This formulation is most easily captured in SETL if we make use of an auxiliary set *path(sb)*, defined to be the set of successors of *interval* which can be reached from sb along a path all but the last node of which is contained within *interval*. In terms of *path*, the SETL formula we require as follows:

```
movable(interval,sint) =
      [+: b(n) ∈ interval] ((posavail(b) - defavail(b)) *
      [+: sb∈cesor(b)|sb=sint(1) or sint∈path(sb)]movable(b,sb));
```

The sets *movable* defined in this way can be computed iteratively within *redproc2* during the second part of the first pass of an overall code-motion/redundancy algorithm (as seen previously, this same pass computes the *posavail* sets using the set *latch* for this purpose). We use reverse interval order, convenient in computing the *path* sets. At each node b we apply the analog of equation (5) above; the set *path(b)* is computed at the same time. In computing *path(b)* we make use of the fact that there exists a path from b to a successor interval *sint* of *interval* if there either is a path from some successor of b to *sint* or if the head of *sint* is a successor of b. Note that *intnodes* is the set of all nodes of interval, and that the map *intov* gives the interval containing a given node.

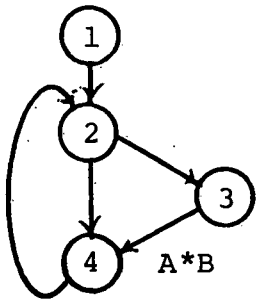
```
/* perform various intializations for interval head, which can be
   successors of later nodes in interval */
sucintervals = cesor(interval);
path(interval(1)) = sucintervals;
/* initialize movable sets */
(∀sint ∈ sucintervals)
   movable(interval,sint) = nℓ;
end ∀sint;
/* pass through interval in reverse order */
(#interval ≥ ∀i ≥ 2) b = interval(i)
/* recompute posavail set */
   posavail(b) = headav * posavail(b) + defavail(b);
/* compute difference for code motion */
   diffmove = posavail(b) - defavail(b);
/* initialize for computation of path set
   path(b) = nℓ;
/* examine each successor of b and determine the set sints of
   all intervals which can be reached from b */
   (∀sb ∈ cesor(b))
      sints = if sb n ∈  intnodes  then {intov(sb)}
                                   else path(sb);
/* update movable  set for each successor interval that can
   be reached */
```

```
      (∀sint ∈ sints)
         movable(interval,sint) =
            movable(interval,sint)+(diffmove*movable(b,sb));
      end ∀sint;
/* update path set */
      path(b) = path(b) + sints;
   end ∀sb;
end ∀i;
```

We will for the moment defer presentation of the entire routine *redproc2* with movability included; before giving a complete algorithm, we must deal with the knotty problem of 'safety'.

iii. Safety of Code Motion

Consider the following loop:



A = some value; A*B

In block 3, variable A is set to some new value and A*B is recomputed. In block 4, A*B is computed also. The methods just outlined would cause the expression A*B to be inserted at the end of block 1 and eliminated from block 4. However, this ignores the possibility that the conditional branch to block 3 shown in the preceding figure was deliberately inserted by a programmer anticipating and wishing to avoid a floating point overflow and an associated interrupt. By simply inserting A*B in block 1 we run the risk of thwarting that purpose. What is therefore needed for a fully acceptable code motion algorithm is a systematic method to determine when it is "safe" to insert an expression at a given point in a program. An expression is *dangerous* if, given improper arguments, it can cause an error interrupt.

is *unsafe* to insert a dangerous computation at a point P where it can cause an error interrupt that might not occur in an untrans-

formed program executed with the same data. More specifically, it is *unsafe* to insert a dangerous expression at a point P if there is a path in the program from P to a program exit which does not contain an upwards-exposed instance of the same expression (with the same arguments). Such a path might allow an error interrupt inevitable in a transformed program to be avoided in the original program.

We will compute *unsafe(b)* -- the set of all expressions which in the sense described above, it is unsafe to insert immediately preceding the block b. A *kill* of an expression is either a redefinition of one of its arguments or a program exit. An expression C is in *unsafe(b)* if there is a *C-clear* path (i.e., a path which contains no instance of C) to a kill of C. The kill may be located in the block b or beyond it.

Let *inskill(b)* be the set of expressions for which there is a clear path to a kill within b. Let *thru(b,sb)*, where sb is a successor of b, be the set of expressions for which there is a clear path through b to sb. An expression is unsafe at the entry of b if it is *inskill(b)* or if it is in *thru(b,sb)* and unsafe at the entry of sb. This relationship is expressed by the following SETL code fragment.

```
(6)  unsafe(b) = inskill(b) + [+: sb ∈ cesor(b)](thru(b,sb)*unsafe(sb))
```

This equation is of exactly the same form as an equation used in the dead variable trace algorithm described previously, and methods like those already used can be employed to compute the *unsafe* sets. The basic idea is this: if we have the *unsafe* sets for an interval and each of its successors, we can compute *unsafe(b)* for each block b in the interval by processing the blocks of the interval in reverse interval order. To apply equation (6) we must have the *unsafe* sets for each of the successors of b. (The sets *inside* and *thru* are initially available for basic blocks and are computed for intervals by a process to be described below.) In computing *unsafe(b)* we deal with a block b and its successor sb; we meet one of three situations:

1. sb can be another node of the same interval I as b (but not the head of I), in which case we will have computed *unsafe(sb)* because we process I in reverse interval order (all successors of a node are treated before the node is treated).
2. sb can be the head of I, in which case *unsafe(b)* is *unsafe(interval)*, because entry to an interval is identical with the entry to its head.
3. sb can be the head of some interval successor *sint* of I, in which case we use *unsafe(sint)* for *unsafe(sb)* (same reason as above).

It follows that equation (6) can always be applied if we process each interval in reverse interval order.

Now we will describe a way of incorporating the safety computation that has just been sketched into the routine *optcess* described a few pages earlier. Let *dangerous* be the set of all dangerous expressions in the program. We will not insert any dangerous computations at any point of which they are unsafe. This restriction leads us to reduce the set *insert(interval)* as follows.

(7) insert(interval) = insert(interval)-(dangerous*unsafe(interval));

This leads to the following final version of the *optcess* algorithm:

```
define  optcess(interval);
/* avail, posavail, defavail, movable, insert, dangerous, inskill,
   thru, unsafe, and cesor  are assumed to be global */
/* compute insert set using equation (4) */
insert(interval) = ([*: sint ∈ cesor(interval)]
                        movable(interval,sint)) -  avail(interval);
/* reduce insert set for safety (cf. equation (7)) */
insert(interval) = insert(interval) -
                                    (dangerous * unsafe(interval));
/* add inserted subexpressions to avail set for interval entry */
newavail = avail(interval) + insert(interval);
/* compute avail and unsafe sets for nodes of interval */
  interval ≥ ∀i ≥ 1) b = interval(i);
/* equation (6) */
```

```
   avail(b) = posavail(b) * newavail + defavail(b);
   unsafe(b) =inskill(b) +
                  [+: sb ∈ cesor(b)] (thru(b,sb) * unsafe(sb));
end ∀i;
end optcess;
```

Next we turn our attention to the changes to *redproc1* that are required to compute the sets *inskill* and *thru* on pass 1 of a final, overall, code motion algorithm. The problem is to compute these sets for an interval when we have them for the nodes of that interval. Let *taux(b)* be the set of subexpressions for which there is a clear path within the interval from the interval entry to the block b. Clearly,

(8) taux(interval(1)) = comps;

since a null path can have no redefinitions on it. For all other nodes b, there is a C-clear path from interval entry to b if there is such a path to some predecessor pb of b and through pb to b. Thus:

(9) taux(b) = [+: pb ∈ predecessors(b)] (taux(pb) * thru(pb,b));

There is a C-clear path through *interval* to one of its successor intervals *sint* if there is a C-clear path to a predecessor pb of *sint* within *interval* and a C-clear path through pb to *sint(1)*. Thus:

(10) thru(interval,sint) =

[+: pb ∈ (pred(sint(1)) * nodes(interval))]

(taux(pb) * thru(pb,sint(1)));

There is a C-clear path to a kill within *interval* if there is such a path to b (within the interval) and C is in *inskill(b)*.

(11) inskill(interval) =[+:b∈nodes(interval)](taux(b)*inskill(b));

The following final version of a revised version *redproc2* of the previously given *redproc1* includes the computations just displayed. Note once more that *intnodes* is the set of all nodes of an interval, and that the map *intov* gives the interval conta: g a given node.

```
define redproc2(interval);
   cesors, pred, defavail, posavail, comps, nokill, expdown,
   inskill, thru, movable, and path are all assumed to be global */
/* initialize for processing */
   head = interval(1);
defavail(head) = nl;
posavail(head) = comps;
taux(head) = comps;  inskill(interval) = nl;
/* define the set of nodes in the interval */
intnodes = {nds, nds(i) ∈ interval};
/* the first pass to get defavail, inskill(interval), taux, and
   an initial estimate of posavail */
(2 ≤ ∀i ≤ #interval)
   b = interval(i);
   preds = pred(b);
/* equation (1) */
   defavail(b) = [*: pb ∈ preds] transf(pb,b,defavail);
   posavail(b) = [*: pb ∈ preds] transf(pb,b,posavail);
/* equation (9) */
   taux(b) = [+: pb ∈ preds] (taux(pb) * thru(pb,b));
/* equation (11) */
   inskill(interval) = inskill(interval) + taux(b) * inskill(b);
end ∀i;
/* compute the set of expressions actually available on entry
   to head */
headav= [*: pb ∈ (pred(head) * intnodes)]transf(pb,head,posavail);
posavail(head) =  headav;
/* form set of successor intervals */
sucintervals = cesor (interval);
   path(interval(1)) = sucintervals;
/* initialize movable sets */
(∀sint ∈ sucintervals)
   movable(interval,sint) = nl;
end ∀sint;
/* pass through interval in reverse order */
   nterval ≥ ∀i ≥ 2)
   b = interval(i);
```

```
/* recompute posavail set */
   posavail(b) = headav * posavail(b) + defavail(b);
/* compute difference for code motion */
   diffmove = posavail(b) - defavail(b);
/* initialize for computation of 'path' set */
   path(b) = nl;
/* examine each successor of b and determine which intervals
   can be reached (the set sints) */
   (∀sb ∈ cesor(b))
sints = if sb n ∈ intnodes then {intov(sb)} else path(sb);
/* update movable set for each successor interval that can
   be reached */
   (∀sint ∈ sints)
      movable(interval,sint) =
            movable(interval,sint) + (diffmove *movable(b,sb));
   end ∀sint;
/* update path set */
   path(b) = path(b) + sints;
   end ∀sb;
end ∀i;
/* compute expdown, nokill, and thru sets for the interval */
(∀sint ∈ sucintervals)
   preds = pred(sint(1)) * intnodes;
   expdown(interval,sint)=[*:b∈preds]transf(b,sint(1),defavail);
   nokill(interval,sint)=[*:b∈preds]transf(b,sint(1),posavail);
/* equation (10) */
   thru(interval,sint)=[+: b∈preds] (taux(b) * thru(b,sint(1)));
end ∀sint;
end redproc2;
```

All that is left is to redefine the overall driving routine for the redundancy code-motion process. We shall rename this main routine *csxcm* (common expression elimination-code motion). The steps needed to treat code motion are now clear;
to incorporate safety considerations, we must compute the *unsafe* set for the outermost interval. Let *exits* be the set of pseudo-blocks which serve as program exits. At each exit, everything is unsafe, so

(12) (∀e ∈ exits) unsafe(e) = comps;;

Given this initialization, we can compute *unsafe(intervals(#intervals))* using equation (6). The overall code for *csxcm*, whose single parameter is the derived sequence *dseq* of a program graph, is then as follows. (Note that, in accordance with our overall treatment of exit nodes, these nodes are assumed never to be absorbed into any interval; thus they remain in the final graph of *dseq* as the only successors of the unique 'program interval' to which an originally given program is finally reduced.

```
define csxcm(seqd);
/* cesors, avail, comps, inside, thru, exits, and unsafe
    are assumed to be global */
/* inner-to-outer pass */
(∀graph(k)∈seqd|k ge 2)(∀int∈hd graph)redproc2(int);;;
/* nothing available on entry to program */
avail(∋hd seqd(#seqd))= nl;
/* everything unsafe at exits */
(∀e ∈ exits) unsafe(e) = comps;;
/* equation (6) */
progint= ∋hd seqd(#seqd);
unsafe(progint) = inskill(progint)+
    [+: e ∈ cesor(progint)](thru(progint,e) * unsafe(e));
/* outer-to-inner pass to complete optimization */
(#seqd ≥ ∀k ≥ 1)(∀int ∈ hd seqd(k)) optcess(int );;;
end csxcm;
```

This completes our description of the common subexpression elimination code motion algorithm. The SETL algorithm just presented has been programmed for readability rather than efficiency. Various detailed changes improving efficiency can be contemplated. We leave consideration of such improvements to the reader.

6. Operator Strength Reduction*

i. Introduction.

Operator strength reduction, sometimes simply called reduction in strength, is an optimization procedure generalizing optimization by code motion. Typically, strength reduction allows an 'expensive' operation like multiplication to be replaced within a loop by addition, an operation which is ordinarily faster. Similarly, exponentiation may in some cases be replaced by multiplication, and, more generally, calculations which can be performed incrementally replaced by their incremental forms. In the pages which follow we will confine ourselves to a discussion of the reduction to repeated addition of multiplications occurring within a loop. Not wishing to extend the present section to a systematic treatise on optimization techniques, we will give only a simplified account of a central aspect of this potentially complex optimization.

We begin with an example illustrating the significance and utility of strength reduction. Let a programming language (such as FORTRAN or ALGOL) providing arrays of fixed dimension be given, and suppose that within such a language a loop

(A)

```
        ...
        i = 1
label:  a(i,j) = b(i,j)
        i = i+1
        if(i .le. 10) go to label
        ...
```

is used to transfer values from one 10×10 array to another. The code produced by naive compilation of this source (but assuming elimination of redundant calculations) will be roughly

---

* The material in the following pages is adapted from K. Kennedy's SETL Newsletter 102, entitled *Reduction in Strength Using Ha Temporaries*. Like most other things in the technique of code optimization, the basic ideas go back to John Cocke.

(B)

```
        ...
        i = 1
label:  t1 = i-1
        t2 = t1 * 10
        t3 = t2 + j
        a(t3) = b(t3)
        i = i+1
        if(i .le. 10) go to label
        ...
```

A reduction in strength algorithm like that shortly to be described will eliminate the multiplication appearing in the foregoing code, transforming this code into something approximating

(C)

```
        ...
        i = 1
        t_i*10 = i*10
label:  t_1 = i - 1
        t_t_1*10 = t_i*10 - 10
        t2 = t_t_1*10
        t3 = t2 + j
        a(t3) = b(t3)
        i = i + 1
        t_i*10 = t_i*10 + 10
        if (i .le. 10) go to label
        ...
```

In writing the code sequence (C), we have introduced a notation which reveals something of the method to be employed in the algorithms to follow. Specifically, the symbol t_{i*10} is used to represent an auxiliary variable which, at each moment, contains the current value of the expression i*10. Similarly, t_{t_1*10} represents an auxiliary variable which always contains the current value of t_1*10.

A subsequent process, not now to be described, which combines variable names designating the same value, and which eliminates the computation of dead results, can reduce this code to a form which is essentially

(D)

```
        ...
        i = 1
        t_i*10 = i * 10
label:  t_t_1*10 = t_i*10 - 10
        t3 = t_t_1*10 + j
        a(t3) = b(t3)
        i = i + 1
        t_i*10 = t_i*10 + 10
        if (i .le. 10) go to label
        ...
```

Good hand-coding for this same loop would be

(E)

```
        ...
        i = 1
        t4 = i * 10
        t5 = j - 10
        t3 = t4 + t5
label:  a(t3) = b(t3)
        i = i + 1
        t3 = t3 + 10
        if(i .le. 10) go to label
        ...
```

or, still better,

(F)

```
        ...
        i = 1
        t4 = 10
        t5 = j - 10
        t6 = j + 90
        t3 = t4 + t5
label:  a(t3) = b(t3)
        t3 = t3 + 10
        if (t3 .le. t6) go to label
        ...
```

Note, however, that whereas the transformation of (D) into (E) can be accomplished by an extension of the methods which we are about to explain, transformations like that carrying (E) into (F) require the use of optimization techniques which we shall not discuss at the present time (linear text replacement).

On many large computers the loop in (F) will run 75% faster than the code shown in (A).

In applying reduction in strength, we consider some strongly connected subregion *scr* of a program graph (as, for example, the strongly connected subpart of an interval). A quantity is a region constant (relative to *scr*) if its value never changes within *scr*. Region constants can be detected (and their computations moved out of *scr*) by the code motion algorithms described in the preceding section. In what follows, we shall assume that this has been done; we shall denote region constants (relative to an assumed region *scr*) by the symbols $c, c_1, \ldots$ etc. As already noted, the idea of reduction in strength is to replace each multiplication

```
x = i * c
```

occurring in *scr* by an assignment

```
x = t
```

re t is a temporary which holds the current value of i * c over ... entire region *scr*. We shall in general denote the temporary

introduced for this purpose by

$$t_{i*c} ;$$

thus t_{i*c} will contain the value of i*c.

To use this approach we must do two things to assure that t_{i*c} always contains the correct value.

1) An initialization of the form $t_{i*c} \leftarrow i*c$ must be inserted just prior to entry to *scr*. For this purpose, it is useful to assume that each strongly connected region has a *prolog* -- a basic block which is always executed just prior to entering the region. New initializations will be inserted at the end of the prolog.
2) After each instruction which modifies i we must insert an instruction which adjusts the value of t_{i*c} appropriately. This implies some slight complications since instructions of the following forms can occur.

(1)

Instruction	Operation to be Inserted
$i \leftarrow c_2$	$t_{i*c} \leftarrow t_{c_2*c}$
$i \leftarrow -c_2$	$t_{i*c} \leftarrow -t_{c_2*c}$
$i \leftarrow j + c_2$	$t_{i*c} \leftarrow t_{j*c} + t_{c_2*c}$
$i \leftarrow j - c_2$	$t_{i*c} \leftarrow t_{j*c} - t_{c_2*c}$

This table shows that we must not only create temporaries for i*c but also for j*c and c_2*c for every j and c_2 that can affect the value of i. In addition, we must insert initializations and modifications for all those temporaries. In what follows, we will describe a systematic procedure for doing this.

ii. Representation of the code to be Optimized.

Since to optimize by reduction in strength we must involve ourselves with code details finer than those which have yet been considered, we shall now outline a hypothetical form of intermediate code representing the text to be optimized. We schematize this code, i.e. represent only its relevant aspects. Generally speaking, we take intermediate code to be a set of SETL blank atoms (each corresponding to one elementary instruction). A function *next* structures this collection of nodes as a list; a function *op* associates an instruction vector with each node.

Let *at* be an atom of code.

a. *op(at)* is the operation code for the instruction. The value *op(at)* is a tuple, generally giving the input variables and output variables for an operation, but in some cases (and epsecially for transfers and conditional transfers) giving somewhat different information. The kinds of operation tuples provided, and the specific forms assumed for them, are as follows.

Operation Type	Form of Corresponding Tuple
No-op	<noop-opcode>
Halt	<halt-opcode>
Add	<add-opcode, out-var, in-var_1, in-var_2>
Subtract	<sub-opcode, out-var, in-var_1, in-var_2>
Negate (x = -y)	<neg-opcode, out-var, in-var>
Transfer (x =y)	<sto-opcode, out-var,in-var>
Multiply	<mult-opcode,out-var,in-var_1,in-var_2>
Function-call	<fcall-opcode,out-var,in-var_1,..., in-var_n>
Subroutine-call	<scall-opcode, in-var_1,...,in-var_n>
Indexed load	<ixl-opcode, out-var,array-name,index-var>
Indexed store	<ixs-opcode, array-name, index-var, in-var>
Branch	<branch-opcode>
Branch conditionally	<cbranch-opcode, in-var>

b. The function *next* maps each node *n* representing a non-branch instruction into its successor instruction in the body of intermediate code being processed. If *n* represents a conditional transfer, then next(n) is a pair <n1,n2>, the two components of this pair being the two possible successor nodes.

c. The operation code associated with an operation is always the first component of the tuple which describes the operation. The macro

```
macro opn(at) = op(at)(1) endm;
```

retrieves this code.

d. The standard 'target variable' for an operation (such as an add, a store, or a function call) which produces an output will in each case be the second component of the vector *op(at)*. Hence, the macro

```
macro targ(at) = op(at)(2) endm;
```

retrieves this variable.

e. The two arguments of operations like *add* and *mult* are retrieved by the macros

```
macro arg1(at) = op(at)(3) endm;
macro arg2(at) = op(at)(4) endm;
```

The first of these macros can also be used to retrieve the unique argument of the operations *neg* and *sto*.

f. When applied to a node *at*, the auxiliary function *args* returns a tuple whose components are the input arguments of the operation represented by *at*. Code for this very simple function will not be given explicitly.

Note that "store" (sto) moves the value of its argument to the target. "Negate" (neg) negates the value of its argument before moving it to the target. "Add" (add) adds its two input operands and places the result in its target variable; "subtract" (sub) and "multiply" (mult) have equally obvious actions.

Our algorithms will occasionally insert instructions into code contained within a strongly-connected program region, so we will need a subroutine to insert an instruction after another instruction. We will never insert code immediately after a branch instruction, so the complicated flow problems which such insertion would raise need not be dealt with.

The insertion routine *insert* shown below has a rather trivial structure. It inserts an operation *optupl* in *codeset* at a position immediately following the node *nd*.

```
define insert(optupl, nd, codeset);
/* op, next  are assumed to be global */
/* get new blank atom  and insert it into the code set */
node = newat;  codeset = codeset with node;
/* attach operation to new node */
op(node) = optupl;
/* make node the successor of nd */
<next(node), next(nd)> = <next(nd), node>;
return;
end insert;
```

iii. Finding Induction Variables.

One of the first tasks of the strength reduction process is to locate the induction variables appearing in a strongly-connected program region. In general, induction variables are those variables which are defined in terms of region constants and other induction variables by operations of the following form:

$$x \leftarrow \pm\ y$$

$$x \leftarrow y \pm z$$

where y and z are either region constants or induction variables. To locate induction variables a process of elimination is used. Specifically, we use the following scheme for finding variables which are not induction variables. Let *indvars* designate the set of all induction variables and let *rconsts* designate the set of region constants for our strongly-connected region.

i. If x is the target of an operation whose code is not one of {neg,add,sub,sto}, then x is not in *indvars*.

ii. If there exists a subroutine call instruction *s* in the strongly-connected region such that $x \in args(s)$, then x is not an induction variable. This restriction removes from *indvars* all variables which could be modified by a subroutine side effect.

iii. If x is the target of any operation one of whose inputs is not in *indvars* + *rconsts*, then x is not in *indvars*.

The algorithm for finding induction variables proceeds by passing through the nodes of the set *scr* (the strongly-connected region), and by collecting all variables which are targets of {add,sub,sto,neg} into a set *indvars* while collecting all subroutine arguments into a set *subargs*. The difference between these two sets gives an initial approximation to the set of induction variables. We then pass through the code repeatedly applying restriction (iii) until no more variables can be eliminated from *indvars*. The routine *findivars* is coded in SETL as follows.

```
definef findivars(scr,rconsts);
   op is assumed to be global */
/* scr is the set of nodes of the strongly-connected region,
   rconsts is the set of region constants,
   indvars is the set of induction variables,
   subargs is the set of variables which are arguments to subroutines,
   ivnodes is the set of instruction nodes which make
        assignments to variables in indvars */
<indvars, subargs, ivnodes> =<nl,nl,nl>;
/* pass through the region applying rules  (i) and (ii) to get
   an  initial approximation for  indvars */
(∀n ∈ scr)
   if opn(n) ∈ {add,sub,sto,neg} then
        indvars = indvars with targ(n);
        ivnodes = ivnodes with n;
   else if opn(n) eq scall then /* rule (ii) */
        subargs=subargs+{(args(n))(i), 1≤i≤#args(n)};
   else if opn(n) ∈ {mult, fcall, ixs,ixl} then /* rule (i) */
        subargs = subargs + targ(n);
   end if;
end ∀n;
/* take the difference to form the approximation */
indvars = indvars - subargs;
ivnodes = {n ∈ ivnodes | targ(n) ∈ indvars};
/* we can now restrict our attention to ivnodes -- the set
   of instructions which modify potential induction variables.
   we pass through ivnodes, eliminating induction variables
   which do not obey restriction(iii)*/
oldiv = nl;
(while indvars ne oldiv) oldiv = indvars;
   (∀n ∈ ivnodes|arg1(n) n ∈ (indvars+rconsts) or
                                   arg2(n) n ∈ (indvars + rconsts))
        indvars = indvars less targ(n);
   end ∀n;
/* reduce ivnodes */
   ivnodes = {n ∈ ivnodes | targ(n) ∈ indvars}
end while;
```

```
return <indvars,ivnodes>;
end findivars;
```

Note that the value returned by the function *findivars* is a pair, consisting of the set of all induction variables and the set of all instructions which modify those variables.

iv. Finding Candidates for Reduction

The algorithm given below aims to reduce all multiplications of the form

$$i * c$$

where i is an induction variable and c is a region constant. These multiplications can be found by passing through the region and checking the arguments of all multiplications. The following routine *findcands* returns the set of nodes which represent operations of appropriate form having appropriate arguments.

```
definef findcands(scr,rconsts,indvars)
/* op is assumed to be global */
/* scr is the region, rconsts is the set of region constants,
   indvars is the set of induction variables */
/* initialize */
cands = nl;
/* pass through scr looking at multiplications */
(∀at ∈ scr | opn(at) eq mult)
   if arg1(at) ∈ indvars and arg2(at) ∈ rconsts then
            cands = cands with at;
   else if arg2(at)∈indvars and arg1(at)∈rconsts then
        /* switch arguments to establish canonical form */
        <arg1(at),arg2(at)> = <arg2(at),arg1(at)>;
        cands = cands with at;
   end if;
end ∀at;
return cands;
end findcands;
```

v. The Temporary Table.

The idea of reduction in strength is to replace

$$x = i * c$$

by

$$x = t$$

where t is a temporary which holds the current value of i * c over the entire region. A variable t_{i*c} will be generated for each multiplication operator which we reduce; this temporary contains the value of i * c. Initializations and modifications must be inserted for these temporaries. The various cases which will arise are sketched on page 315.

To handle all the cases which can arise we shall use a routine which computes a set *affect* of ordered pairs

$$\langle i,x \rangle$$

Here, i is an induction variable and x is an induction variable or region constant which appears in an assignment to i. This routine makes a pass through the scr building an initial *affect* relation from instructions which appear in assignments to induction variables. Additional items are then filled in using a transitive closure procedure.

```
definef findaffect(ivnodes,indvars,rconsts);
/* op is assumed to be global */
/* ivnodes is the set of nodes which set  induction varibles,
   indvars is the set of induction variables, rconsts is the set
   of region constants */
/* initiailize so that each induction variable affects itself */
affect = {<x,x>, x ∈ indvars};
/* pass through ivnodes to get the initial affect relation -- any
   operands of an instruction which sets x must affect x */
(∀at ∈ ivnodes) x = targ(at);
   if pair args(at) then
        affect{x} = affect{x} + {<x,arg1(at)>,<x,arg2(at)>};
   else
        affect{x} = affect{x} with <x,arg1(at)>;
   end if;
end ∀at;
/* now take the transitive closure: to affect{x}, add any
   variable which affects an induction variable in affect{x} */
n = 0;
(while #affect gt n) n = #affect;
   (∀x ∈ indvars)
        affect{x} = affect{x}+ {<x,y>, y∈affect[indvars*affect{x}]};
   end ∀x;
end while;
return affect;
end findaffect;
```

Once we have the *affect* set we can accomplish strength reduction very neatly using the temporary table. If i*c is a candidate for reduction, we must form

$$t_{x*c} \text{ for all } x \in affect\{i\}$$

inserting appropriate initializations and modifications for these temporaries.

In the SETL routine which follows we shall use a mapping t(x,y) which maps x and y to the unique compiler-generated name for t_{x*y}. The initialization instruction required for each temporary will be inserted at the end of the prolog when the entry for that temporary is inserted in the table. This will require a pointer *plast* to the last instruction in the prolog.

```
define  streduce(prolog,plast,scr,rconsts);
/* op  is assumed to be global */
/* prolog is the initialization block whose last instruction is
   plast, scr  is the region, and rconsts are the region constants
   which we assume are found in an earlier code-motion pass */
/* find induction variables */
<indvars, ivnodes> = findivars(scr,rconsts);
/* find candidates for reduction */
cands = findcands(scr,rconsts,indvars);
/* calculate the 'affect' relation */
affect = findaffect(ivnodes,indvars,rconsts);
/* now pass through the candidates creating temporaries and
   inserting initializations and modifications */
(∀at ∈ cands) x = arg1(at);  c = arg2(at);
/* create new temporaries as required */
   (∀y ∈ affect{x} | t(y,c) = Ω)
        t(y,c) = newtemp( );
        /* the function newtemp,  for which code will not be
           shown, is assumed to return a compiler generated
           variable name */
/* insert initialization instruction in prolog */
```

```
      insert(plast,<mult,t(y,c),y,c>,prolog);
   /* avoid double entries in the const*const case */
      if y ∈ rconsts then t(c,y) = t(y,c);;
   /* insert modifications to the new temporaries after
      instructions  which set induction  variables*/
      (∀n ∈ ivnodes | targ(n) eq y)
           newargs = if pair args(n) then <t(arg1(n),c),t(arg2(n),c)>
                                 else <t(arg1(n),c)>;
   /* the inserted instruction has the target t(y,c), involves
      the same operation as n, and has newargs  as its argument */
           insert(n,<opn(n),t(y,c)> + newargs,scr);
      end ∀n;
   end ∀y;
   /* now replace the candidate operation entry by a store operation*/
   op(at) = <sto, targ(at), t(x,c)>;
 end ∀at;
end streduce;
```

This completes the presentation of our strength reduction algorithm.

It is appropriate to note two of the limitations of this algorithm as presented.

(1) The algorithm does not include a systematic clean-up of the code; i.e. it generates more temporaries than are minimally necessary and does not subsequently eliminate them.

(2) The algorithm does not recognize the fact that generated temporaries are themselves induction variables.

It is possible to develop more complex strength reduction algorithms in which these shortcomings are removed; however, we shall not do so now.

7. Some Packing Algorithms Useful in Register Assignment.

Packing algorithms are used in optimizers to assign quantities to registers in an effective way. We shall conclude this section by giving two algorithms of this kind. In each case, we assume that the quantities to be assigned form a set s on which a map *inter* is defined; if x ∈ inter(y), then x and y are "interfering" quantities and cannot be assigned to the same register. The algorithms to be described will establish a map *rep* on s; each y ∈ s will then be assigned the same register as *rep*(y), so that only those x with rep(x) eq Ω need be assigned registers to begin with, and the other assignments will follow automatically. The number of registers required is then equal to #{x∈s|rep(x) eq Ω}; we wish to reduce this quantity to a minimum. Naturally, if rep(x) and rep(y) are to be the same, x and y must be non-interfering.

It should be noted that the problem we shall discuss is formally identical with that of coloring the nodes of a graph in such a way that no two adjacent nodes have the same color. The minimal number of colors required is the so-called *chromatic number* of the graph. The problem of finding a minimum coloring is now known to belong to an extensive class of mutually equivalent problems, all of which are strongly suspected of requiring exponentially many steps (in terms of their given data) for exact solution. The considerably more efficient algorithms which will be presented in the following pages are consequently algorithms which give a good, but not always the best, packing.

The first algorithm, due to J. Cocke, begins by arranging the quantities x in decreasing order of #inter(x). The first quantity x in sequence is then removed, and the remaining quantities scanned in sequence to find quantities y for which rep(y) may be assigned as x without violating the interference condition. Whenever rep(y) is defined, y is removed from the sequence. This process repeats as often as necessary, as shown in the following SETL algorithm, wherein we presume that the sequence *seq* of quantities x arranged in decreasing order of #inter(x) has been pre-calculated.

```
rep = nl;  low = 0;
(while low < ∃j ≤ #seq|rep(seq(j)  is x)eq Ω doing low = j;)
    rep(x) = x;
    interfere = inter(x);  k = j;
    (while k < ∃kk ≤ #seq| n(seq(kk) is y ∈ interfere)
        and rep(y) eq Ω doing k = kk;)
        interfere = interfere + inter(y);
        rep(y) = x;
        end while k;
end while low;
```

Suppose in the above algorithm that #inter(seq(j)) = n_j. It is clear that, if seq(j) is not assigned a representative before the k-th iteration of the outer while-loop in the above algorithm, then seq(j) interferes with at least k quantities preceding it in sequence; hence $k \leq \min(j-1, n_j)$. The maximum possible value of k, which is plainly also the maximum number of "registers" needed for the packing by the above algorithm of all quantities, is therefore at most

$$\max_j \min (j, n_j+1) . \tag{1}$$

If $j_1 > j_2$ and $n_{j_1} > n_{j_2}$, we clearly reduce (1), or at any rate fail to increase it, if we interchange n_{j_1} and n_{j_2}. With a given pattern of interferences, (1) therefore takes on its smallest possible value if, before applying the SETL procedure shown above, we arrange the elements in decreasing order of number of interferences We have presupposed such an arrangement. Suppose now that k is the value of the maximum (1). Then, for $j \leq k$, $n_j > k-1$; hence the total number n of interferences is at least k(k-1). Reading this relationship backwards, we see that Cocke's algorithm allows a collection of registers to be divided into no more than $\sqrt{n}$ groups of mutually noninterfering registers if the total number of interregister interferences is n. The worst case is that in which there exists a set s of approximately $\sqrt{n}$ registers each of which interferes with all the other registers of s.

This line of thought suggests a more expensive but more powerful :king algorithm, suggested to the author by A. P. Ershov. Suppose that in assigning representatives we decide to assign x and y the same representative, thus in effect identifying x and y. The interferences of the resulting 'identification element' are then inter(x) + inter(y). If we then choose y so that #(inter(x) * inter(y)) is as large as possible, the largest possible number of interferences will be eliminated by the identification of x and y. A SETL algorithm expresssing the procedure suggested by this observation is as follows:

```
/* quants is the total collection of quantities being considered;
    curq those to which no representatives have yet been assigned;
    curint the current state of intereferences, taking the assign-
    ment of representatives into account; we write y ∈ repdby(x)
    if, as far as our calculation yet shows, y is to be represented
    by x */
curq = quants;  curint = inter;  repdby = {<x,{x}>, x∈quants};
/* determine pair to identify, if any */

loop: intno = 0; /* initialize intersection number; then search
    for noninterfering  pair x,y with maximum number of common
    interferences */
    (∀x ∈ curq) /*form the set of 'neighbors once removed' of x */
        naybsofnaybs=curq*[+:z∈curq*curint(x)]curint(z)-curint(x);
        (∀y ∈ naybsofnaybs)
            if #(curq*curint(x)*curint(y)) is n gt intno then
                intno = n;  bestpair = <x,y>;
            end if;
        end ∀y;
    end ∀x;
```

```
    if intno eq 0 then
    /* see if any pair of noninterfering elements exists */
        if n ∃ x ∈ curq, y∈curq|n y∈curint(x) then go to done;;
        /* else prepare to identify x with y */
        bestpair = <x,y>;
    end if intno;
    <x,y> = bestpair; /* and now identify x with y */
    repdby(x) = repdby(x) + repdby(y);
    curint(x) = curint(x) + curint(y);
    (∀z ∈ curq * curint(y)) curint(z) = curint(z) with x;;
    go to loop;
done: rep = {<y,x>, x ∈ curq, y ∈ repdby(x)|y ne x};
```

Item 17. ADDITIONAL FEATURES OF THE SETL LANGUAGE.

In the pages which follow, we will extend our account of SETL by describing several additional linguistic features which it provides. This will bring our account of the abstract language to completeness, except in regard to its extendability features, specifically the system of user-defined object types and object-type dependent operators which SETL provides. This important feature of the language will be discussed in a later section. Note also that the SETL data structure elaboration language (DSEL) is significant for the overall SETL concept, even though the DSEL is efficiency-oriented rather than expressivity oriented, and in this sense not part of the abstract SETL language. Discussion of the DSEL is postponed to Item 20 below.

1. Supplementary discussion of generalized assignments.

The generalized treatment of assignment statements presented in Item 13 above let us make any programmed function p(x) available for use on the left-hand side of an assignment statement by associating a procedure opstore(x,y) of one additional parameter with it. The procedure *opstore* and the function *op* must stand in an abstract 'storage-retrieval' relationship to each other. This requires that in the sequence

```
y = op(x); opstore(x,y);
```

the second operation must be equivalent to a no-operation, etc. We shall now explore the issues relevant to this relationship in greater detail than we did in Item 13, in this way completing our discussion of the 'sinister call' facilities of SETL. Specifically, we shall present a family of mechanisms which allow storage-retrieval associations to be made explicit.

In some cases, the storage operation corresponding to a retrieval can be deduced from the form of the latter. In other cases, this may be entirely impossible, since a storage operation may set in motion some special train of actions (such as a call to a ind-the-scenes 'allocate' function) impossible to anticipate

merely from the form of the associated retrieval operation. In either case, however, it is apt to be convenient to keep the storage and the retrieval code together, as a good deal of this code, if not all of it, is likely to be common to both situations. The syntactic conventions described in the following paragraphs are intended to conform to these principles.

Note that, in the system to be described, code representing both the storage and the retrieval operations associated with a user defined function f occurs within a single SETL function body i.e., between a function-opener line

(1) definef f(x);

and the closing line

(2) end f;

which corresponds to (1). The new basic features of the system to be described are load and store blocks. The relevant syntax is as follows:

A. Explicit store and load blocks. A user-defined function which performs a retrieval operation is allowed to contain explicit *store blocks* and *load blocks*. The form of a load block is

(3) (load) block;

a load block may also be terminated in any of the styles

(4) (load) block end;
(load) block end load; etc.

The form of a store block is

(5) (store name) block;

which may also be terminated as

(6) (store name) block end;
(store name) block end name; etc.

Here *name* is any legitimate SETL variable name. Code not belonging either to a store or a load block, which is of course allowed also, is called normal code.

Whenever a function is called, a flag signifying whether the 1 is a dexter or a sinister call will be transmitted. Normal code will

be executed in any case. *Store blocks* will be bypassed if the call is dexter; *load blocks* bypassed if the call is sinister.

B. Return statements within load blocks, store blocks, and normal code. The form of a return statement within a load block will be that customary for return statements within a function. The form of a return statement within a store block will be of the form appropriate for a subroutine return. Within a store block (5), *name* will designate the quantity to be stored, which is available as an implicit argument of a sinister call, transmitted when a sinister call to a function is initiated, but appearing by syntactic convention in the store block header rather than among the arguments listed in the function header.

Return statements in normal code will have the form appropriate for function returns, but will be expanded in a somewhat unconventional fashion. Each statement

(7) return expn;

occurring in a normal code section will be syntactically analyzed and expanded into two blocks of code, the first being a load block, the second being a store block. The load block corresponding to (7) has simply the form

(8) (load) return expn;;

The store block corresponding to (7) involves a more elaborate transformation of (7). We explain this in 'top down' fashion.

i. The store block corresponding to the normal-code statement (7) has the form

(9) (store name) block; return;;

where *block* is a code sequence determined, in a manner to be explained, by the syntactic form of the *expn* occurring in (7). We call *block* the *corresponding code* of *expn*.

ii. If *expn* has the conditional form

(10) if $cond_1$ then $expn_1$ else if $cond_2$ then $expn_2$ ... else $expn_k$

then its corresponding code is

) if $cond_1$ then act_1 else if $cond_2$ then act_2 ... else act_k ;

where act_j is the corresponding code of $expn_j$.

iii. If expn is a call on a programmer-defined prefix function and thus has the form

(12) $$f(expn_1,\ldots,expn_k)\ ,$$

then the code corresponding to (12) is that obtained by expanding the sinister call

(13) $$f(expn_1,\ldots,expn_k) = name;$$

here *name* is the variable appearing in the header of the store block in which (12) is imbedded.

iv. Remark (iii) applies also to programmer-defined functions of infix or prefix form, and also to programmer-defined functions with zero arguments. We extend it also to all those primitive SETL operations for which standard retrieval operators are provided. These are the following.

$$\underline{hd}\ a,\quad \underline{t\ell}\ a,\quad a(x),\quad a\{x\},\quad a[x],\quad t(i:j),$$

and various related multi-parameter operations; together with $<a_1,\ldots,a_n>$, whose sinister correspondent is the SETL multiple assignment.

v. If the principal operator in an expression is a primitive SETL operation which is not known to the SETL system as a retrieval, or is a constant, the expression is <u>disqualified</u>. The code corresponding to a disqualified expression is

```
noop;
```

vi. If the principal operator in an expression is an existential or universal quantifier, then the expression is disqualified. Note however that, as shown in detail below, storage sequences could in some circumstances be assigned even to expressions of this kind.

vii. If an expression consists of a code block, i.e., has the form [; block], it is treated as if it were a call on a function $f(p_1,\ldots,p_n)$, f being defined by

```
definef f(p1,...,pn);
block
end f;
```

and where $p_1,\ldots,p_n$ is the complete list of variable names appe ing in *block*. (See subsection 3 below for additional detail.)

To give a composite example, we note that the store block corresponding to the return statement

(14) return if a gt 0 then progf(a)[progf(b)] else Ω;

is

```
(15)  (store name) if a gt 0 then t = progf(a); tt = progf(b);
                   t[tt] = name; progf(a) = t;;
```

Even though the standard SETL system will not do so, storage procedures can be associated with a surprisingly large number of SETL operators. Thus many more operators than one might at first imagine can be used in sinister position. As a first example, consider the membership operator

(16) $x \in s$.

A procedure standing in the appropriate 'storage' relationship to the 'retrieval' (16) is

(17) (store b) if b then s=s with x else s=s less x; end if;;

The reader should convince himself that (16) and (17) do indeed form a storage-retrieval pair. This is by no means the only possible way of defining a storage operator corresponding to the retrieval (16). For example, the procedure

```
(18)  (store b) if b then
                   if s eq nl then s = {x};
                                  else x =ɜs;;
              else x = {s};  end if b;;
```

might also be used. This serves to emphasize the fact that a retrieval operator does not determine an associated storage operator uniquely; only consideration of the larger algorithmic context within which a storage-retrieval pair is to be used, and especially consideration of the collection of those other storage operators of which a given operator is to be independent, will make firm the choice of storage operator which is to correspond to a given retrieval.

Nonuniqueness of storage procedure appears even more plainly -- we consider the boolean or operation

(19) x or y .

A storage procedure corresponding to this retrieval is

```
(20)    (store b) if b then
                     if not (x or y) then x = t;;
                  else x = f; y = f; end if;;
```

Clearly, however, we could as well write

```
(21)    (store b) if b then
                     if not (x or y) then y = t;;
                  else x = f; y = f; end if;;
```

The boolean operator

```
(22)                          not x
```

has the storage procedure

```
(23)                 (store b) x = not b;;
```

Once storage procedures for the basic Boolean operators or and not have been chosen the general theory of nested storage-retrieval pairs presented in Item 13 allows storage procedures to be deduced for all the Boolean operators: these follow from the expression of boolean operators in terms of or and not. We might of course write

```
(24)  definef x and y; return not (not x or not y); end and;
      definef x exor y; return x and not y or (y and not x);
                                 end exor;
      definef x implies y; return not x or y; end implies;
```

etc. The reader will verify that, if (24) is used to define and, while (21) is used to define the storage sequence corresponding to or, then in virtue of (24) and our general expansion rules the operation and inherits a storage sequence equivalent to the following:

```
(25)              (store b) if not b then
                               if x and y then y = f;;
                            else x = t; y = t; end if;;
```

Similarly, the operation implies inherits the storage sequence

```
(26)              (store b) if b then
                               if not (x implies y) then y = t;;
                            else y = f; x = t; end if;;
```

Various set-theoretic operations such as intersection and union e also retrievals. Storage sequences (which are generally nonunique) may be assigned as follows:

(27)
```
for x * y (intersection):
(store z) y = y-x+z; x = x+z;;
```

(28)
```
for x + y (union):
(store z) x = x*z; y = y*z+(z-(x+y));;
```

(29)
```
for x - y (set-theoretic difference):
(store z) y = y-z; x = x*y+z;;
```

(30)
```
for x//y (symmetric difference):
(store z) x = x*(z+y) + (z-(x+y)); y=y*(z//x);
```

Storage sequences may be assigned to various important comparison and arithmetic operators. We have

(31)
```
for x ge y:
(store b) if b ne (x ge y) then <x,y> = <y,x>;
                end if;;
```

Since the operators le, lt and gt can be expressed using ge and not, storage sequences for the remaining arithmetic comparisons can readily be worked out. We have also

(32)
```
for x max y:
(store z) if x lt z then y =  z;
                else if y lt z then x = z;
                     else x = z; y = z; end if y;;;
```

(33)
```
for abs(x):
(store z) x = sign(x) * z;;
```

(34)
```
for -x
(store z) x = -z;;
```

Since min can be expressed in terms of x max y and -x, a storage sequence for min can readily be deduced. For the arithmetic addition, subtraction, and division operators the following storage sequences are available:

)
```
for x + y:
(store z) y = z-x;;
```

(36)
```
for x - y:
(store z) y = x-z;;
```

(37)
```
for x/y:
(store z) x = y*z;;
```

Note that (36) is deducible from (35) and (34). This list of storage sequences can be extended. For example, the integer arithmetic operation x//y (reduction of x modulo y), the real multiplication operator, the 'integer part' function, and various other similar operators all may be assigned storage sequences.

Next we consider the quite surprising case of the universal quantifier. If C(x) is an expression involving the free parameter x, then the quantified expression

(38) $$\forall x \in a \mid C(x)$$

may be assigned the following storage sequence:

(39)
```
(store b) if b then
  (while ∃x ∈ a|not C(x)) C(x)=t;;
else if (∀x ∈ a) C(x) then
  C(∋a) = f; end if;;
```

Naively put, this storage sequence, when called upon to store the value t, corrects any violation of the condition C(x) eq t which is found, and iterates until all violations are corrected. (Of course, an infinite loop may be implied.) A similar but more complex storage sequence can be assigned to more elaborate universally quantified expressions such as:

(40) $\forall x_1 \in e_1, \min(x_1) \leq x_2 \leq \max(x_1), \ldots \mid C(x_1,\ldots,x_n)$.

Since de Morgan's well known law of duality allows the existential quantifier

(41) $$\exists x \in a \mid C(x)$$

to be expressed in terms of (38) and the boolean not operation, a storage sequence for (41) can readily be deduced. This is

(42)
```
(store b) if not b then
   (while ∃x ∈ a | C(x)) C(x) = f;;
     else if not ∃x ∈ a | C(x) then
        C(∋a) = t;  end if;;
```

In a suggestive number of cases, the combination of (39) with our general expansion rules generates sequences which force a desired condition merely from the assignment of t to an appropriate expression representing the condition. This points, albeit from afar off, to a 'prescriptive' as opposed to a 'procedural' style of programming. For example, the statement

(43) $(\forall x \in s \mid f(x) \in s) = t;$

that the set s should be transitively closed under the mapping f expands into the sequence

(44) $(\text{while } \exists x \in s \mid f(x) \text{ } n \in s) \; f(x) \text{ } in \; s;;$

i.e., into a transitive closure routine. Similarly, the statement

(45) $(1 \leq \forall n < \#tup \mid tup(j) \text{ } le \text{ } tup(j+1)) = t;$

expands into

(46) $(\text{while } 1 \leq \exists j < \#tup \mid tup(j) \text{ } gt \text{ } tup(j+1)) \langle tup(j), tup(j+1) \rangle = \langle tup(j+1), tup(j) \rangle;;$

i.e., into a version of the bubble sort. Other, still more complex procedures are seen to result from the expansion of sinister calls whose left-hand side is a condition to be 'forced'. It must be admitted, however, that until an attack is made on the substantial optimization problems which the suggested approach raises, the practicality of a prescriptive approach to programming like that which has just been described remains small.

The generalized treatment of assignment operations which SETL embodies rests on the abstract notion of storage/retrieval relationship defined by the conditions of Item 13 (page 25) above. It deserves to be noted that this relationship is merely the simplest of a variety of somewhat more general relationships which one is accustomed to exploit in programming. Among other similarly used concept-pairs are the pair "stacking-operator/unstacking-operator" and the pair "enqueing-operator/dequeueing-operator". We may, for example, call a monadic operator op an unstacking operator if there exists a corresponding procedure opstack(x,y) such that

(47) (i) if op x is repeatedly evaluated after a sequence of calls opstack(x,y_1);...;opstack(x,y_n); then the sequence of values obtained will be $y_n, y_{n-1}, \ldots, y_1$;

(ii) the operation

opstack(x, op x);

(i.e., unstacking followed by immediate re-stacking) is equivalent to a no-operation.

Relationships of this type, while more complex and less universally useful than the fundamental retrieval-storage relationship which has been discussed at length above, are nevertheless worth noting and exploiting in many situations. The SETL sinister call facilities which we have outlined can be used for this purpose; note that the use of explicit load and storage blocks provides very considerable flexibility.

For example, an enqueue-dequeue pair may be programmed as follows.

```
(48)  definef queue q;
      (load) if #q ne 0 then x=q(1); q = q(2:);
                return x; else return Ω; end if;;
      (store x) q(#q+1) = x;;
                end queue;
```

Using this function, elements may be enqueued and retrieved from a queue simply by writing

(49) queue q = x; and queue q ,

respectively. Queues involving more general disciplines, as for example priority queues, may be handled in much the same way.

The treatment of stacks is just as simple. We can define a stacking operator by writing

```
(50)  definef topoff stack;
      (load) if(#stack) gt 0 then x=stack(#stack);
                stack(#stack) = Ω; return x;
                else return Ω; end if;;
      (store x) stack(#stack+1) = x;;
      end topoff;
```

called as

(51) topoff stack = x .

This operation places x on a stack. If used in a dexter call, it returns the top element of the stack, at the same time removing it from the stack.

Still more general operators of this same type can be useful in connection with other data structures. Suppose for example that we take a binary tree to be represented by a triple <node,ℓ,r>, where ℓ is the left descendant function in the tree and r is the right descendant function in the tree. The following function will permit a new element x to be hung at the lowest-leftmost element below *node* simply by writing

leftchild tree = x;

The required definition is simply as follows:

```
definef leftchild tree; <node,ℓ,r> = tree;
(while ℓ(node) ne Ω) node = ℓ(node);
return tree(2)(node);
end leftchild;
```

Note also that the left-hand use of recursive functions will sometimes prove useful when inherently recursive data structures such as trees are to be treated.

2. An extended example of the use of generalized assignments.

The following extended example will show the manner in which multiple sinister calls may be used. We consider a hypothetical (systems-) programming situation in which names will be encountered, and in which each name may have one of a substantial number of attributes. In the 'typical' case, however, most attributes of most names will be undefined. We suppose for definiteness that all the attributes of a name are character strings. In what follows we shall describe a treatment of this assumed situation in which all attribute-representing character strings are maintained 'locally' as long as the total number of characters which they involve does not exceed a stated limit. When this total exceeds its allowed limit, however, the attribute-representing character strings are assigned integer identifiers and are moved to a packed storage array.

The preceding describes a moderately elaborate storage mechanism. We emphasize that our general sinister call conventions allow this whole mechanism to be hidden from a programmer using it, who need merely write

(1) j attr name = x;

to assign the j-th attribute of *name*, and use the expression

(2) j attr name

to fetch the value of this attribute. In this fact, and in the circumstance that these same conventions aid conceptually and pragmatically in the realization of storage schemes by clearly defined, modular structures, lies the claimed advantage of our system of sinister assignment. The reader, in scrutinizing the example which follows, can judge the extent to which this claim is justified.

The first layer of the storage scheme which we gave as an example explicitly 'hashes' the parameter *name* appearing in (1) and (2); note that the value of this parameter is a string. The hashtable scheme used is as follows: a comprehensive packed

character array *string* containing a single copy of each name used in conjunction with an array *stack* whose entries are tuples

(3) <chain, firstchar, length, attribute entry> .

In (3), *chain* links together all entries with a common initial hash reference, finally becoming zero; *firstchar* is the index in *string* of the first character of the name belonging to the item (3), and *length* is the length of this name; *attribute entry* contains the attribute information for the name. The entry *chain* in (3) is -1 for empty entries in *stack*, which might of course be examined in the first step of a hash search.

```
definef m attr name;
/* fetches and stores the m-th attribute of a given name */
/* we assume a function hash which computes an initial trial
   entry in stack for a given string */
/* the 'packed character array' string is assumed to be global,
   and will be used in several routines appearing later in the
   present subsection */
macro chain, firstchar, length, attentry; endm;
loc = hash name;
(while chain(stack(loc)) gt 0 doing loc = chain(stack(loc));)
   if chain(stack(loc)) lt 0 then quit;; /* since empty slot */
   if length(stack(loc)) is len ne #name  then continue;
   else if string(firstchar(stack(loc)):len) ne name then
        continue;
   else return m att attentry(stack(loc));
   end if;
/* the operator att defines the next layer of the present
   storage system  */
end while;
/* on fall thru loop we may deal with a name being encountered
   for the first time */
```

```
if chain(stack(loc)) eq 0 then
   chain(stack(loc)) = #stack+1;
   loc = #stack+1;
end if;
stack(loc) = <0, #string+1, #name, initial( )>;
/*  here  the function initial is assumed to return the 'initial
   setting' of an attribute entry, corresponding to the 'all
   attributes undefined' condition */
string = string + name;
return m att attentry(stack(loc));
end attr;
```

We must now describe the next innermost storage routine att. At this level of our storage system, various details hidden at the preceding level become visible. As already stated, the attribute values stored in an attribute entry will be character strings. If they all fit, these values are stored directly within a locally maintained string *attrstring*. If not, they are stored within the global 'packed character array' *string* mentioned in the attr routine; in this case, *attrstring* stores integers (decimally encoded as strings) which indicate substring starting positions within *string*. Each attribute entry contains a flag called *lflag*, which indicates whether attribute entries or integers referencing them are found within the *attrstring* portion of the entry. To enhance the efficiency with which our hypothetical storage system handles undefined attributes (and we imagine most attributes of most objects to be undefined) we provide each entry with a group of flags, equal in number to the maximum possible number of attributes.

An attribute entry is therefore a triple

<attrstring,lflag,deflags>

consisting of a character string, a 1-bit flag, and a boolean string. Accordingly, the function *initial* called by the attr routine is as follows:

```
definef initial;
  turn <nulc,f,nats*0b>;
end initial;
```

here *nats* is the total number of attributes which could possibly be associated with an entry.

The code for att is as follows.

```
definef j att entry;
macro attrstring, lflag, deflags; endm;
/* the 'undefined' attribute value, which we assume is
   represented by the null character, is treated specially.
   the insertion of a value for a hitherto undefined
   attribute is signalled by transmission of a negative
   parameter to the lower-level attrval routine */
negflag = 1; /* standardize flag controlling transmission of
   negative parametrs */
(load)
   /* return nulc for undefined attribute */
  if n deflag(entry)(j) then return nulc;;
end load;
(store atval)
   if atval eq nulc then
        /* no-op if attribute is already undefined */
        if n deflag(entry)(j) then
            return;
        else /* one of the deflag flags must be dropped */
            deflag(entry)(j) = f;
        end if;
   else if n deflag(entry)(j) then
        /* a value is being set up for a hitherto undefined
              attribute */
        negflag = -1;
        deflag(entry)(j) = t;
   end if atval;
   | store;
```

```
/* now calculate the number of defined attributes preceding the
   j-th attribute, in order to get the appropriate 'internal
   address' of an item to be retrieved (or stored) */
ndefd = #{k, 1 ≤ k < j | deflag(entry)(k)} + 1;
/* now fetch or store from the appropriate part of the
   attstring portion of entry, noting the case in which
   a value is to be set up for a hitherto undefined attribute */
return (negflag * ndefd) attrval entry;
end att;
```

The attribute store/fetch operator attrval reduces to one or the other of directattr, codedattr, depending on the setting of the *lflag* in *entry*.

```
definef j attrval entry;
macro lflag(x); x(2) endm;
return if lflag(entry) then
                        j codedattr entry
       else             j directattr entry;
end attrval;
```

At this point we must specify still more of the inner structure of the *attstring* portion of an attribute entry, structure which will of course be reflected in the routines codedattr and directattr just encountered. We shall suppose *attrstring* to be a character string within which there occur blocks of information delimited by blanks. For dealing with a 'blank delimited string memory' of this kind, we introduce the following, relatively primitive, substring retrieval-insertion operator, called word. It retrieves or modifies the j-th word in such a string; if j is negative, it inserts a new word between the former (j-1)st and j-th positions.

```
definef j word cstring;
   scan to the (j-1)st blank, and locate the j-th blank, if any */
startloc = 0;
jp = if j lt 0 then -j else j;
(1 ≤ ∀k < jp)
   must = startloc < ∃n ≤ #cstring| cstring(n) eq ' ';
   startloc = n;
end ∀k;
if n startloc < ∃n ≤ #cstring| cstring(n) eq ' ' then
   n = #cstring+ 1;;
(load) return cstring(startloc+1: n-startloc-1);;
(store substr)
   if j lt 0 then /* a new word is being inserted */

        if jp eq 1 then /* make insertion in first position */
            cstring = substr + ' ' + cstring;
        else if n eq #cstring+1 then /* insertion in last position*/
            cstring = cstring + ' ' + substr;
        else /* insertion in some middle position */
            cstring = cstring(1:startloc) + substr
                          + ' ' + cstring(startloc+1:);
        end if jp;
        return;
   end if j;
   /* here we treat the case in which an old word is being modified*/
   if substr eq nulc then /* an old word is being abolished */
        cstring = cstring(1:startloc) + if n lt #cstring
                           then cstring(n+1:) else nulc;
   else /* an old word is being modified */
        cstring = cstring(1:startloc)+substr
                  + if n le #cstring then cstring(n:) else nulc;
   end if;
   return;
end store;
end word;
```

Next we give code for the directattr storage-retrieval function. If called for retrieval, j directattr entry merely returns the j-th blank-delimited word of the *attstring* portion of *entry*. The consequences of a sinister call may be more complex, since upon such a call overflow requiring restructuring of the contents of *attrstring* may occur.

```
definef j directattr entry;
macro attrstring(x); x(1)   endm;
(load) return j word attrstring(entry);;
(store atstr)
   if j lt 0 then /* an entirely new entry is being inserted */
        lenchange = #atstr + 1;
   else if atstr ne nulc then /* an old entry is being changed */
        lenchange = #atstr   - #(j word attrstring(entry));
   else /* an attribute is being dropped.
                                use nominal lenchange value */
        lenchange = -1;
   end if;
   if(#attstring(entry)+ lenchange) lt maxchars /* maxchars is the*/
        maximum number of characters which will be accommodated
        locally */
        then /* simply modify attribute string */
            j word attrstring(entry) = atstr;
   else /* an overflow occurs. change from directattr
        to codedattr representation */
        oldattrstring= attrstring(entry);
        k = 1; attrstring(entry)=nulc; lflag(entry)=t;
        (while((k word oldattrstring(entry)) is  kth attr ne nulc
                                doing k = k+1;)
            k codedattr entry =  kth attr;
        end while;
        /* now modify old attribute or insert new attribute */
        j codedattr entry = atstr;
   end if;
   return;
end store;
end directattr;
```

Next we represent the codedattr routine in SETL, which volves making still further details explicit. This routine will fetch and retrieve attributes, but proceed indirectly using integer 'addresses' represented as character strings. These addresses are assumed to reference substrings of the globally available packed character array *string* , which made its appearance in the first algorithm, attr, presented in the present section. Each 'address' is assumed to be a decimal coded, blank terminated, integer referencing a starting character position in *string*; the attribute field which begins at this position is assumed to be terminated by a blank. We will assume that it is impossible for an entry to have so many defined attributes that one cannot store all these attribute addresses locally.

The storage part of the procedure shown below is somewhat more complex than the retrieval part, since a kind of overflow can occur on storage. Note that in the following routine *string* is assumed to be a quite long character string, and that we structure the code so as to avoid shifting long substrings of *string*.

```
definef j codedattr entry;
/* string is assumed to be global */
if j gt 0 then /* locate start of old attribute string */
   charaddr = dec (j word attrstring(entry));
   /* locate blank which terminates attribute-representing string*/
   must = charaddr ≤ ∃m ≤ # string|string(m) eq ' ';
end if;
(load) return string(charaddr: m-charaddr);;
(store atstr)
   if j lt 0 then /* a new attribute is being inserted */
        j word attrstring(entry) = dec (#string + 1);
        string = string + atstr+' ';  return;
   end if;
   /* else an old attribute is being modified */
   /* first determine amount of trailing blank space */
   if n m ≤ ∃nonblkloc≤ #string|string(nonblkloc) ne ' ' then
        nonblkloc = #string+1;
```

```
   avail = nonblkloc - charaddr-1;  /*total available space
                                    for entry */
   if (#atstr) le avail then
        string(charaddr:#atstr)= atstr;
   else /* place the attribute-representing string at the
                                    very end of string */
        string(charaddr:avail) = avail*' ';
        j word attrstring(entry) = #string + 1;
        string = string + atstr + ' ';
   end if;
   return;
end store;
end codedattr;
```

2 Code-blocks within expressions. Inverted function, subroutine, and macro definitions.

SETL allows any block of value-returning code to be used within an expression. The syntactic form provided is

(1) [; functionbody] .

Here, *functionbody* designates any block of code which could validly be the body of a defined function, i.e., which could appear in the context

(2)
```
definef fcn(p1,...,pk);
functionbody
end fcn;
```

Note that *functionbody* will therefore contain statements of the form

(3) return e;

e being an expression. The value of e at the moment at which a statement (3) is executed defines the value of the expression (1). Note therefore that an occurrence of (1) in an expression is precisely equivalent to an occurrence of the call

(1') $fcn(p_1,\ldots,p_n)$,

where $p_1,\ldots,p_n$ is a list of *all* the variables occurring in the *functionbody* (1).

A variety of inverted constructions related to (1) are also provided. In the first place, these include

(4a) [; *functionbody* definef fname($p_1,\ldots,p_n$);]

and

(4b) [; *macrobody* macro mname($p_1,\ldots,p_n$);]

These forms are provided in response to the psychological fact that one generally recognizes the desirability of making a given code sequence available as a function or macro only after one has explicitly written a first instance of the sequence. The form (4a) is precisely equivalent to an occurrence of the function call

(5a) fname(p_1,...,p_n) ,

where also the function definition

(6a)
```
definef fname(p1,...,pn);
functionbody;
end fname;
```

appears in the same namescope as (4a). Similarly, the form (4b) is precisely equivalent to an occurrence of the macro invocation

(5b) mname(p_1,...,p_n);

where also the macro definition

(6b)
```
macro mname(p1,...,pj);
macrobody;
endm mname;
```

appears in the same namescope as (4b).

To make a sequence of statements available as a subroutine, the form

(7) [; *statementblock* define subname(p_1,...,p_n);]

is provided. This is precisely equivalent to an occurrence of the subroutine call

(8) subname(p_1,...,p_n);

where *subname* is defined by

(9)
```
define subname(p1,...,pn);
statementblock;
return;
end subname;
```

Occasionally one will find it convenient to define a subroutine, function, or macro in this inverted form, but to invoke it for the first time with parameters other than those appearing in the definition (for example, some of theparameters in the first invocation may be constant). To this end, the forms

(10a) [; *functionbody* definef fname(p_1,...,p_n); q_1,...,q_n]

(10b) [; *macrobody* macro mname(p_1,...,p_n); q_1,...,q_n]

(10c) [; *statementblock*; define subname(p_1,...,p_n); q_1,...,q_n]

are provided. Here, for example, (10a) is equivalent to an occurrence of the function call

(11) fname(q_1,...,q_n) ,

where also the function definition (6a) appears in the same namescope as (11). Similar remarks, which the reader will readily deduce, apply in the case of (10b) and (10c).

Note also that inverted macrodefinitions of either the form (4b) or the form (10b) may involve not only ordinary macro arguments but also generalized arguments.

4. Additional discussion of the SETL namescoping facility: the *alias* declaration.

In the present section, we shall complete our account of the SETL namescoping scheme. Information concerning the most basic and generally useful namescoping facilities provided by SETL was given in section 6 of Item 14 above. These facilities, whose description the reader is asked to review, may be summarized as follows:

a. A system of nested scopes is provided. Scope boundaries are logical 'brackets' having the power to protect names occurring within them from identification with names of the same power occurring outside. However, such identification will occur if an item *i* occurring outside a namescope *ns* is 'transmitted to' or 'made known within' *ns*, and if an item with a corresponding name occurs within *ns*.

b. Items *i* can be transmitted between namescopes by use of either global or include declarations. A global declaration typically makes an item *i* known within most or all of the namescopes *ms* physically imbedded within the namescope *ns* within which *i* is made global. An include declaration makes *i* known within the scope within which the declaration occurs. The include declaration also provides a mechanism allowing an item *i* having one name in the scope *ns* within which it first occurs to be known under another name within a scope *ms* to which it is subsequently propagated.

c. Each item *i* which becomes known in a namescope *ns* is referenced there by some particular *compound token*

(1) $$\text{name}_1\_\text{name}_2\_ \cdots \_\text{name}_k$$

consisting of simple names separated by underbar symbols. The same item may also be referenced by using any initial subpart of the compound token (1). Two items *i*, *i*' both known in the same namescope are identified if one is referenced by a name constituting an initial part of the full compound name (1) which could be used to designate the other.

The propagation rules summarized above (and discussed in more tail in Section 6 of Item 14, in which section various restrictions on the use of the global and include propagation mechanisms were also stated) are basic to our namescoping scheme. These restrictions are intended to elminate errors which might easily and inadvertently creep in if over-free use of these declarations were allowed. We shall now discuss a particular one of these restrictions, and present a mechanism which allows it to be relaxed.

Specifically: By making distinct items i_1, i_2 etc. occurring within the proper text of one namescope *ns* available within another namescope *ms*, our namescoping scheme allows i_1 and i_2 to be identified with items j_1, j_2 occurring within *ms*, and then recursively with items occurring within a third namescope *ms'*, etc. We have required that no identification made in consequence of the transmission of the distinct items i_1, i_2 between namescopes should lead to the identification of i_1 and i_2 within *ns*. The use of an explicit alias declaration, in a manner which we shall now describe, gives us a way around this restriction.

The form of an alias declaration is

(2) alias $namelist_1, \ldots, namelist_k$;

or simply

(3) alias *namelist*;

The form of each *namelist* occurring in (1) or (2) is

(4) $(name_1, name_2, \ldots, name_k)$.

If a declaration of the form (2) or (3) appears in a namescope *ns*, then all the names appearing in each *namelist* are understood to be synonyms for the same item. Thus, for example, having reference to the text (17) presented in Section 6 of Item 14 as a sample of SETL code containing a namescoping error, we see that this text may be corrected by the inclusion of an alias statement. When corrected, it will read as follows.

(5)

```
scope routl;
alias (u,v);
... u = 0; v = 1; ...
end routl;
scope rout; global y, routl;
scope rout2;
include u_routl[y]; ...
...
end rout2;
scope rout3;
include v_routl[y];
...
end rout3;
end rout;
```

As remarked, this text becomes illegal if the alias statement appearing in its second line is deleted.

ITEM 18. Internal Specification of the SETL primitives *

1. Introduction. Design issues and decisions.

In the pages which follow, we shall use SETL to give an extended, and quite detailed, specification of the 'run-time library' of routines which realize the SETL primitives in our current implementation. This extends, and carries to a much higher degree of detail, the information given previously as Item 6, SETL Implementation and Optimization.

Before launching into the many details which will be required, let us summarize the main design issues and decisions which these details will reflect. (Cf. also Item 6.) SETL programs may be expected to make heavy use of the following operations.

(a) membership testing;
(b) iteration over sets;
(c) tuple operations, i.e., extraction and insertion of indexable components;
(d) addition of elements to, and deletion of elements from, sets;
(e) functional evaluation, in one of the three forms f(x), f{x}, or f[x], f being a set;
(f) multiparameter functional evaluations, in the appropriate multiparameter generalizations of one of the three preceding forms;
(g) the sinister or storage operations corresponding to the retrievals (e) and (f).

In addition, one should note the fact that SETL, as a value language, will often have to force copying of the compound objects occurring in programs. If the amount of copying forced is not to become excessive, some mechanism for detecting and bypassing logically redundant copy operations is required.

The main design decisions to which these reflections lead have been summarized above in Item 6. The entire implementation makes use of a garbage-collected heap area, a supplementary stack being used for argument passing and recursion control. The implementation

---

* The material which follows is very largely the work of Mr. Henry Warren.

presupposes that sets will be sparsely populated, i.e., that many more objects will be generated, placed in sets, and removed from them than are typically present in a set at any one time. This supposition, together with a desire to represent sets in a form which also supports fast functional evaluation if these sets happen to be sets of pairs, is taken to exclude straightforward representation of sets by bit-vectors, an approach that might otherwise be used if we assume some overall assignment of serial numbers to the elements of sets. Accordingly, in the present SETL implementation, fast membership testing is secured by maintaining sets as tables within which elements are accessed in terms of a calculated hash.

The most vexing, and perhaps questionable, design decision in the present implementation reflects an attempt to ensure that both membership testing and functional evaluation will be performed efficiently in typical cases. Evaluation of f(x) or f{x} requires fast access to the set of all n-tuples with given first component; for this reason, tuples are located within sets by the hashes of their first components, and within a set the collection of all tuples of length $n \geq 3$ with given first component c is always collected together, the set S_c of all tails of such tuples being referenced explicitly by an entry in the hash table representing f. Ordinarily, tuples are represented by arrays of contiguous entries, some growth space generally being reserved. However, the representation of a tuple within a set is different, and much more list- than array-like (details will be given below). This is one of the weak points of the current implementation. Insertion of tuples of length ≥ 3 into sets, extraction of such a tuple from a set by the ∋ operator, and iteration over a set of tuples all involve conversions that can be very time consuming if long tuples are involved. A design which explicitly reflected the fact that maps of more than two parameters are rarely used would probably have been less objectionable in this regard.

Operations such as with or f(x) = y; which require the calculation of 'slightly modified' versions of compound and possibly very bulky data items are implemented using routines which cha these data items rather than reconstructing them completely.

In the final SETL system, these routines will be used when live-
ld analysis assisted by dynamic reference counting shows that no copying operation is necessary; in other cases a copying operation will be forced. In the preliminary SETLB system an explicit COPY primitive is provided and automatic copying is never performed. The great extent to which algorithms are insensitive to this variation of semantics is surprising.

Equality testing is executed very frequently during typical SETL runs. This operation will be optimized as follows. Object types will be compared by a fast in-line test, and a FALSE response given if types differ. Equality of short objects will be determined by fast in-line code. If neither of these fast in-line checks is decisive, transfer will be made to the library equality-test routine. This routine will normally be able to establish inequality quite rapidly (using a readily accessible 'number of elements' or 'number of components', and also a precalculated hash value stored with each set). However, establishing the equality of compound objects will require systematic checking of every element or component of one against the other, and can be a much slower operation.

Certain commonly occurring SETL equality checks, such as x eq Ω or s eq nl , will be treated as special cases and compiled to wholly in-line code.

Some of the most significant routines in the run time library are recursive. It is interesting to observe that, while some of this recursiveness is an artifact resulting from the manner in which sets of n-tuples are handled, another part is inherent in the axioms of set theory. Indeed, one of the axioms of set theory states that two sets are equal if and only if each element of either is equal to some element of the other, so that recursion enters into the very notion of set equality. It may also be remarked that the specifications which follow make it clear that a rather complex program library is required to support the semantically transparent primitives of set theory in an efficient way. This seems to us to support the guess that those primitives are an appropriate
rting point for the construction of a highly expressive programming language.

An issue always significant in the semantic definition of a language is the manner in which calling sequences are handled, which is to say, the rules and conventions which allow the integration, into composite procedures, of individual calls upon semantic primitives. This important matter will be treated, not here, but in subsequent pages. Thus the specification to be given now represents only that part of the semantics of SETL directly associated with its data objects and with the primitive operations which combine these data objects. In our specification, the block of code corresponding to each of the SETL primitives is shown as a SETL function (or subroutine). For example, the code corresponding to the primitive invoked by the source-SETL $f\{x_1,\ldots,x_n\}$ is shown as a routine *ofan(f,x)*, whose first argument is the set (or procedure) to be applied functionally, and whose second argument is an n-tuple x whose n components correspond to the n parameters $x_1,\ldots,x_n$ of the form $f(x_1,\ldots,x_n)$. This style of specification presupposes the existence of a parameter-passing mechanism which gathers together the arguments $f,x_1,\ldots,x_n$ to be supplied to the library routine *ofan*, and passes them along to the *ofan* routine. Note that in the actual implementation the second group $x_1,\ldots,x_n$ of parameters will be gathered into a vector-like, implementation level, object (as for example a block of contiguous positions on a recursion stack), but not necessarily into the fully-fleshed out system of words and fields which represents a SETL vector. As stated, all details of the parameter-passing mechanism will be elided here and given later. In particular, our specifications following will not show how SETL recursions, function calls, parameter modification and returns, or control transfers are actually implemented.

2. SETL object representations at the machine level and at the level of this specification.

A SETL object is represented in a machine, and in our actual implementation (written, as noted in Item 6, in the low level systems-writing language LITTLE) as a series of words that are broken up into fields. In the present specification,

--- do not descend to quite so low a level of description, but stead represent SETL objects (including sets) as n-tuples whose various components show the information associated with the object in the actual implementation. It may be remarked that the main purpose of this specification is to depict the more complex and novel algorithms in the run time library, particularly those dealing with sets. To keep the description at a reasonably high level, many fine details of the run time library as coded in LITTLE have been omitted.

This omission applies in particular to most storage allocation and garbage-collector related details; which is to say that in the present specification we assume a garbage-collected universe of vectors, integers, and strings, and show how sets can be introduced as an additional form of compound object. Our specification does show the expansion and contraction of the hash tables used to represent sets, and the management of the extra 'growth space' with which SETL tuples are provided.

The actual LITTLE-written implementation library distinguishes (in order to avoid wasting space) between 'long' and 'short' items, which in effect adds five data types (short integer, short boolean string, etc.) to the system of types visible to the SETL user, which are also those which will play a role in this specification. These distinctions will however not be made here, and all the detail which these distinctions imply will be omitted. Moreover, none of the inner details involved in the manipulation of the SETL atomic types (such as character strings and bit strings) are shown in this specification. For example, in specifying the concatenation of two strings we completely ignore all necessary iterations over words and characters and all the details of packing and unpacking which are necessary in the actual library implementation. This operation is specified, in a way actually assuming the existence of the SETL concatenation operator, as 's1+s2'. We aim by proceeding in this fashion to emphasize the most interesting aspects of the run time library, and to leave its more mundane details out of the way. Similarly, in describing the ıcatenation of tuples, we show the manner in which the length and 'reserved growth space' fields of a new tuple are formed,

and also show the tests which determine if a concatenated tuple can be formed in space already reserved contiguous to the body of the first of two tuples to be concatenated. However, the actual step of concatenating the two tuple bodies is merely represented as a calculation of 't1+t2', which of course elides a data-move loop present in the actual LITTLE-written implementation of the concatenation operation.

The true run time library includes multiprecision arithmetic routines for dealing with the indefinitely large integers which SETL provides. These routines are not shown here; it may be remarked that multiprecision algorithms of the type required are exhaustively discussed in the second volume of D. Knuth's celebrated treatise.

Recursion is awkward to code in LITTLE, and the execution of a recursive call is substantially slower than that of a normal call. Therefore it is advantageous, in an efficiency-oriented package like the run-time library, to eliminate recursion whenever practical. This is done in the present specification. There are a number of routines (such as those for inserting and removing tuples from sets) which are coded here without recursion, even though they would have been simpler if recursion were used. These routines generally perform some stacking operations, and accordingly routines 'push' and 'pop' are provided (the LITTLE run time library employs the run time recursion-control stack as an auxiliary stack).

The intentions outlined above lead us to use the following conventions in the specification which we are about to give.

Every SETL object is represented by a pair (2-tuple) of the form:

<type, value>,

the type is an integer whose value is (symbolically) *int*, *real*, *bool*, *char*, *blank*, *label*, *sub*, *fun*, *undef*, *tuple*, or *set*. There is also an auxiliary type called 'special pair' whose use is discussed below.

The value field is:

1. The value itself if the object is an atom;
2. A triple of the form

<length, space, tuple>

if the object is a tuple:

3. A 5-tuple of the form

<#members,hashtable load,hashcode,hashtable size, hashtable>

if the object is a set: and

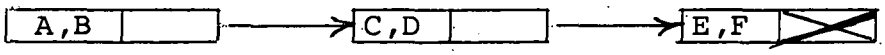
4. A pair of objects if the object is a special pair.

Here *length* is the length of the tuple (greatest index of its defined components), *space* is the space allocation for the tuple (the number of components that could be contained without reallocating), and *tuple* is the tuple itself.

For sets, *#members* is the number of members, *hashtable load* is the load on the set's hash table, *hashcode* is the hash code of the set itself, *hashtable size* is the size of the set's hash table, and *hashtable* is the hash table itself (a tuple whose components are indexed by a hash code).

Each entry in a set-representing hash table is either undefined (absent) or is a list of the set members that occupy the same slot of the hash table (due to hash code clashing).

In this specification, chaining of list elements is represented by nesting of tuples. For example, the list which might be represented diagramatically by



is actually represented by

<A,B,<C,D,<E,F>>> .

As noted above, SETL objects are shown in this specification as pairs whose components are object type and object value respectively.

Accordingly, an object that is a member of a set may be represented either by a pair or by a triple, depending on whethe it has a successor in the list of items sharing a hash entry with it.

We give some examples showing the representation, in this specification, of various typical objects. The integer 2 is represented as

<0, 2> .

The value *true* (a bit string of length one) is represented by

<2, 1B> .

A character string is represented by a pair such as

<3, 'abcde'> ;

the null tuple by the pair

<9, <0, 0, < >>> .

The tuple whose components are the above three atoms would be represented as follows:

<9, <3, 4, <<0,2>, <2,1B>, <3,'abcde'>>>>

The null set has the following representation:

<10, <0, 0, 12345, 0, < >>>

(The null set's hash code of 12345 is arbitrarily chosen.)
A set containing the above three atoms would be represented as follows in the conventions used in the present specification:

```
<10, <3, 3, 27182, 4, <Ω,
                       <3,'abcde'>,
                       <0,2,<2,1B>>,
                        Ω>>>
```

In writing the above representation we have supposed that the hash table size required for our set of three elements is four, that the string 'abcde' has a hash code whose last two bits are 01, and that 2 and 1B both have hash codes ending in 10. This leaves slots 0 and 3 (00 and 11) of the hash table empty, as shown by the pr[illegible]ce (in the hashtable-representing vector) of two undefined components. The set's hash code of 27182 is entirely ficticious.

As remarked above, a nest of triples culminating in a pair used to represent a list of several set members all sharing one single hash table entry.

The hashtable load component of a set is the number of items attached to the hash table. It is used to determine when a hash table should be expanded or contracted. For many sets, hash table load is equal to the number of members. However, for sets containing tuples of length three and up, hashtable load may be considerably less than the number of members. This is because of the fact that when tuples of size three and larger are added to a set they are broken up in a manner that facilitates locating subcollections of such tuples all of which have the same first few components (this is necessary for fast functional evaluation in the general case).

When a tuple of length three or more is put in a set, a special pair is formed whose two components are the tuple's first component and a set containing the tuple's tail.

More specifically, the hashtable entries for a set S are lists containing:

1. non-tuple members of S,
2. zero-, one-, and two-tuple members of S, and
3. special pairs of the form <a,sa>, where a is the first component of an n-tuple ($n \geq 3$) member of s, and sa is a set containing the tails of those n-tuples that begin with a.

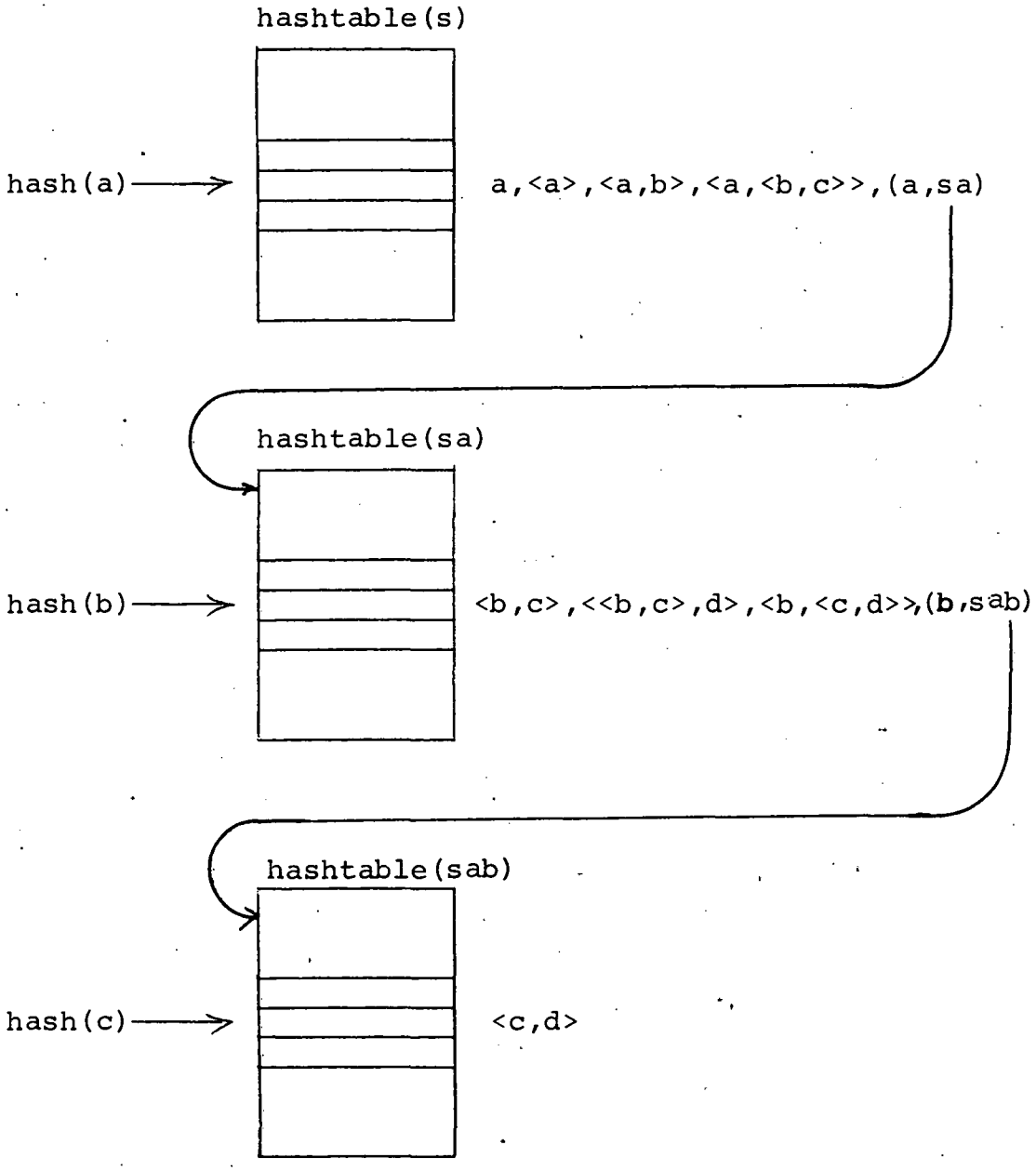
To put a tuple <a,x,y,...,z> of length greater than 3 into a set in whose representation a special pair (a,sa) is already present, one puts its tail <x,y,...,z> into sa, a process which may involve additional, recursive, decomposition of <x,y,...,z>.

We illustrate the scheme of representation which results from this convention by considering as an example the set

s = { a,<a>,<a,b>,<a,<b,c>>,<a,b,c>,<a,<b,c>,d>,
<a,b,<c,d>>,<a,b,c,d>},

in which a,b,c and d are arbitrary objects.

The internal representation of this set is shown in the following diagram.



Showing still more detail, and in particular showing values for the 'number of members' and 'hashtable load' parameters always associated with the internal description of a set, we can represent these same sets by the three following hashtables (shown as tuples).

```
s = <10,<8,5,67890,4,<Ω,
                      Ω,
                      a,<a>,<a,b>,<a,<b,c>>,(a,sa),
                      Ω>>>;

sa = <10,<4,4,13579,4,<Ω,
                      <b,c>,<<b,c>,d>,<b,<c,d>>,(b,sab),
                      Ω,
                      Ω >>>;
```

and

```
sab  = <10,<1,1,24680,1,<<c,d>>>> .
```

To evaluate <a,b> for the above set, the set s must be searched for tuples of the following forms:

```
<a,b,c,...>,
<a,b,<c,...>>,
<a,<b,c,...>>,
<a,<b,<c,...>>>.
```

but not for a tuple of the form <a,<b,c>,...>. This stems from the fundamental SETL definition of functional evaluation for more than one argument:

```
f{x,y} = (f{x}){y} .
```

It would be intolerably inefficient for the run time library to work directly from this definition, as that would require building the intermediate set f{x}, which might very well be quite large.

The tuples of the desired form are not all found in the same place. Those that are pairs (of the forms <a,<b,c,...>> and <a,<b,<c,...>>>) are found in the primary hashtable, the triples (of the form <a,b,<c,...>>) are found in the secondary hashtable, and those of greater length (<a,b,c,...>) are found deeper in the structure.

The breaking up of n-tuples when they are put into sets is a considerable complication to the algorithms (with, less, arbitrary element of, iteration, and functional luation) that deal with sets. These algorithms are probably ce as large in terms of lines

of code as they would otherwise be, and several times more complicated conceptually. The procedure used is predicated on the assumption that a set of n-tuples is probably going to be used as a function, and furthermore that the usual way to build a set-function of n variables is as a set of (n+1)-tuples (i.e., tuples such as <a,b,c,d> rather than <a,<b,<c,d>>>, or some such thing). In particular, the assignment

```
f(a,b,c) = d;
```

causes the 4-tuple <a,b,c,d> to be added to the set f.

3. Special conventions concerning Ω.

One of the purposes of this specification (though not its main purpose is to define the meaning of SETL operators in detail, particularly in those marginal cases which escape attention in broad-brush accounts of operation semantics. Many particularly elusive marginal cases of this sort involve the usage of the 'undefined atom' Ω. Principal facts concerning these cases are summarized here for quick reference.

The rules concerning Ω are usually derived from the following principle. If a set or tuple is referenced in a way that requests a member having certain properties, and no such member exists, then the value of the expression is Ω. On the other hand, execution is generally terminated if a SETL operation is called with incorrect data types, or with 'obviously' incorrect values (including Ω), such as a zero divisor.

Thus the main sources of Ω are those cases of certain retrieval operations (f(x), arb, s, hd t, tupl(n), string(n)) in which the item to be retrieved is not defined (i.e., does not exist).

An essential exception to the above is that Ω is allowed to enter into the eq and ne operations, so that it may be tested for. As a convenience, Ω is also allowed to be the argument of the SETL type, atom, and pair functions, since the use of these indicates that there is some doubt about the status of a SETL object. Again as a convenience, Ω is allowed in hd, tl, and assignment operations.

Ω is allowed to be a component of a tuple, but it is not allowed to be a member of a set. As a component of a tuple, it serves as a 'place marker'. Let

t1 = <1,Ω,3,Ω>, and
t2 = <1,Ω,3>.

Then: #t1= #t2 = 3;
t1 eq t2 is true;
t1(0) = t1(2) = t1(4) = t1(5) = Ω;
t1(4:) = t1(5:) = Ω; and
t1(4:2) = t1(5:2) = Ω.

Arguments are not checked unless they are actually used. For example, in x → s , if s is nl, a value of false is returned even if x is Ω. Similarly, in f{x1,...,xn}, if no tuple in f begins with x1, a nl result is returned and x2,x3,...,xn are not examined for legality. This is a result of the way sets of tuples are implemented, and may be considered to be a departure from 'true' SETL ('true' SETL is not at this time defined to quite such a degree of precision).

Ω is generated or propagated by:

1. x = Ω; (assignment)
2. ∋nl, ∋nulb, ∋nulc, ∋nult
3. hd Ω, hd < >, hd <Ω,...>
4. tl Ω, tl < > , tl <x>
5. random nl
6. tupl(n) if n is nonpositive, or greater than #tupl, or if the n-th component is missing (undefined)
7. string(n) if n is nonpositive or greater than #string
8. f(x), if x n∈ hd [f] (but f(Ω) and Ω(x) are invalid)
9. ∃x → s ↑ c(x), if no such x is found
10. [op: x∈s] expr, if s is nl , nulb, nulc, or nult
11. type Ω
12. f[nl], f[nulb], f[nulc], f[nult]
13. NPOW(n,S), n > #S

Ω behaves as follows, where 'invalid' causes termination of execution:

1. atom Ω is true
2. #Ω is invalid
3. Ω eq Ω is true; Ω ne Ω is false
4. Ω ∈ s and Ω n∈ s are invalid
5. x ∈ Ω and x n∈ Ω are invalid
6. Ω with x and s with Ω are invalid
7. Ω less x and s less Ω are invalid
8. s lesf Ω is valid, and tuples beginning with Ω (sparse tuples) are deleted
9. ∋Ω is invalid
10. Ω{x} and f{Ω} are invalid
11. Ω{xl,...,xn} and f{xl,...,Ω,...,xn} are invalid (although the latter is valid if the result is nℓ based on the arguments preceding the undefined one).
12. f(x) and f[x] are similar to (10) and (11) above
13. pair Ω is false

4. Keypunch conventions used in the specification

In order to allow it to be kept current with the actual run time library, the specification which will be reproduced below is maintained as a computer file. As the full SETL character set is not available for computer output, lexical conventions essentially equivalent to those of SETLB are used in the specification text. These conventions are as follows:

(SETL characters above and SETLB characters below)

1.

{	}	'	[	]	|	∈	Ω	∋	∀
≤	≥	≠	[	]	↑	→	OM.	ARB.	∨

2.

∃	≤	≥	<	>	#
=	≤	≥	<	>	↓

SETL underlining, as in nℓ, is represented in SETLB by an affixed dot, as in NL. .

5. Table of Contents, Index of Routines in this Specification, with an Account of Call-Caller Relationships.

a. Table of contents

b. Index of routines in this specification, with call-caller relationships.

Section	Name	Description
6.1.2	type(x)	type code of SETL object x
	value(x)	value of object x
	nextm(x)	next member after x in a set-list
	ncomps(t)	number of components in tuple t
	space(t)	space allocation existing for tuple t
	tup(t)	actual tuple part of tuple t
	nmembs(s)	number of members of set s
	load(s)	loading of set s
	hcode(s)	hashcode of set s
	htsize(s)	hash table size of set s
	hashtable(s)	hash table of set s
6.1.4	ncompallo(n)	initial space allocation for a tuple
	toodense(s)	true iff hash table of s is overused
	toosparse(s)	true iff hash table of s is underused
6.1.6	initm(x,s)	gets first member of s in set-list (hash table entry) for x
	search(x,s,m)	a macro: aids in searching a set-list
6.1.9	push(x)	stacks x
	pop(x)	pops x
6.2.2	stlread(f,ℓ)	parser for reading input
6.2.3	printer(o,f,m)	main routine for printing
	printc(o,ℓ,f,m)	prints an object o with label ℓ
	charout(o,ℓ,f,m)	converts an object o to a character string, and prints it
	post(f,x)	prints an arbitrarily long character string x
6.3	errimp(m,x)	error exit for impossible type code
	errtype(m,x)	error exit for invalid type code
	errval(m,x)	error exit for invalid data value
	errmsg(m)	error exit, prints message m
	errimpl(m)	error exit for implementation error
	errmix(m,x,y)	error exit for mixed type codes

6.4	hash(x)	calculates the hash code of x
6.5	atom(x)	SETL atom x
6.6	nelt(x)	SETL #x (number of elements)
6.7	equal(x,y)	SETL x eq y, main routine, calls eqbasic, eqtupnt, and subsetnt
	eqtupnt(x,y)	SETL x eq y for non-trivial tuple comparison
	subsetnt(x,y)	SETL x eq y for non-trivial set comparison (same as a subset test)
	eqbasic(x,y)	utility routine used by equal, eqtupnt, and subsetnt
6.8	elmt(x,s)	SETL x ∈ s, main routine, calls elmtbst,elmtcst, elmttup, and elmtset
	elmtbst(x,s)	SETL x ∈ s for s a boolean string
	elmtcst(x,s)	SETL x ∈ s for s a character string
	elmttup(x,t)	SETL x ∈ t for t a tuple
	elmtset(x,s)	SETL x ∈ s for s a set, calls elstup and elssmp
	elssmp(x,s)	SETL x ∈ s for s a set, x not a tuple of length ≥ 3, calls equal
	elstup(x,s)	SETL x ∈ s for s a set, x a tuple, calls equal
6.9	augment(x,s)	SETL s = s with x, main routine, calls augaok
	augaok(x,s)	SETL s = s with x, arguments OK, calls augsimp and augtup
	augsimp(x,s)	SETL s = s with x, x not a tuple of length ≥ 3, calls equal, toodense, and expand
	augtup(x,s)	SETL s = s with x, x a tuple, calls equal, push, pop, augsimp, tupsplt, and replace
	replace(x,s,y)	replaces x in set s by y, used only by augtup and dimtup, calls equal
	tupsplt(t,i,j)	builds up tuple t from component i to j+1, calls setlsmp
	setwth1(x)	SETL {x}, main routine, calls setlsmp and tupsplt
	setlsmp(x)	SETL {x}, x not a tuple of length ≥ 3
	expand(s)	doubles the size of the hashtable of s, calls hash

6.10	diminis(x,s)	SETL s = s less x, main routine, calls diminok
	diminok(x,s)	SETL s = s less x, arguments OK, calls dimsimp and dimtup
	dimsimp(x,s)	SETL s = s less x, x not a tuple of length ≥ 3, calls equal, toosparse, and contrct
	dimtup(x,s)	SETL s = s less x, x a tuple, calls dimsimp, equal, push, pop, and replace
	contrct(s)	halves the size of a hash table
6.11	dimf(x,s)	SETL s = s lesf x, main routine, calls dimfaok
	dimfaok(x,s)	SETL s = s lesf x, arguments OK, calls equal, toosparse, and contrct
6.12	arb(s)	SETL ∃s, main routine, calls arbbstr, arbcstr, arbtup, and arbset
	arbbstr(s)	SETL ∃s, s a boolean string
	arbcstr(s)	SETL ∃s, s a character string
	arbtup(t)	SETL ∃t, t a tuple
	arbset(s)	SETL ∃s, s a set, calls arbsimp, tupaddl, and concatt
	arbsimp(s)	SETL ∃s, s a set, tuples of length ≥ 3 not rebuilt
6.13	next(c,s)	SETL iteration ∀x ∈ s, main routine, calls nextbit, nextchr, nextcmp, and nextmem
	nextbit(c,s)	SETL ∀x ∈ s, s a boolean string
	nextchr(c,s)	SETL ∀x ∈ s, s a character string
	nextcmp(c,t)	SETL ∀x ∈ t, t a tuple
	nextmem(c,s)	SETL ∀x ∈ s, s a set, calls tupaddl and concatt
6.14	of(f,x)	SETL f(x), retrieval, main routine, calls ofbstr, ofcstr, oftuple, and ofset
	ofbstr(f,x)	SETL f(x), retrieval, f a boolean string
	ofcstr(f,x)	SETL f(x), retrieval, f a character string
	oftuple(f,x)	SETL f(x), retrieval, f a tuple
	ofset(f,x)	SETL f(x), retrieval, f a set, calls equal and arbset

	ofn(f,x)	SETL f(x1,...,xn), retrieval, calls ofan and arbset
	ofa(f,x)	SETL f{x}, retrieval, calls equal, augaok, and aunion
	ofan(f,x)	SETL f{x1,...,xn}, retrieval, calls equal, augaok, augsimp, tupsplt, aunion, push, and pop
	ofb(f,s)	SETL f[s], retrieval, main routine, calls ofbbool, ofbchar, ofbtup, and ofbset
	ofbbool(f,s)	SETL f[s], retrieval, s a boolean string, calls ofset, ofbstr, and plus
	ofbchar(f,s)	SETL f[s], retrieval, s a character string, calls ofset, ofcstr, and plus
	ofbtup(f,t)	SETL f[t], retrieval, t a tuple, calls ncompallo and ofset
	ofbset(f,s)	SETL f[s], retrieval, s a set, calls nextmem, aunion, and ofa
	ofbn(f,s)	SETL f[s1,...,sn], retrieval, calls ofbset
6.15	sof(f,x,r)	SETL f(x) = r ('storage of'), main routine, calls sofbstr, sofcstr, softupl, and sofset
	sofbstr(f,x,r)	SETL f(x) = r, f a boolean string, calls okindex
	sofcstr(f,x,r)	SETL f(x) = r, f a character string, calls okindex
	softupl(f,x,r)	SETL f(x) = r, f a tuple, calls okindex
	okindex(x)	verifies that x is a reasonable index
	sofset(f,x,r)	SETL f(x) = r, f a set, calls dimfaok and augsimp
	sofn(f,x,r)	SETL f(x1,...,xn) = r, calls dimfn, tupaddl, and augtup
	sofa(f,x,r)	SETL f{x} = r, calls dimfaok, nextmem, and augsimp
	sofan(f,x,r)	SETL f{x1,...,xn} = r, calls dimfn, space, nextmem, and augtup
	sofb(f,s,r)	SETL f[s] = r, calls nextmem, dimfaok, and augsimp
	sofbn(f,s,r)	SETL f[s1,...,sn] = r, calls nextmem, dimfn, tupaddl, and augtup

	dimfn(f,x)	Alters f by removing a minimum number of members to make f{xl,...,xn} null. Calls equal, push, pop, toosparse, and contrct
6.16	copy(x)	Copies x to its full depth. Calls copy
	copy1(x)	Copies x, but only one level deep
6.17	head(x)	SETL hd x
	tail(x)	SETL tl x, calls ncompallo
6.18	plus(x,y)	SETL x = x + y, main routine, calls aunion and concatt
	concatt(t1,t2)	SETL t1 = t1 + t2, tuples. Calls ncompallo
	aunion(x,y)	SETL x = x + y, sets, calls nextmem and augaok
6.19	minus(x,y)	SETL x = x - y, main routine. Calls setdiff and pminus
	setdiff(x,y)	SETL x = x - y, sets, calls nextmem and dimaok
	pminus(x)	SETL -x (prefix minus)
6.20	mult(x,y,r)	SETL r = x*y, main routine, calls intsect
	intsect(x,y,r)	SETL r = x*y, sets, calls nextmem, elmtset, and augaok
6.21	divide(x,y,r)	SETL x = x/y and r = x//y, main routine. Calls symdiff
	symdiff(x,y)	SETL x = x/y, sets (symmetric difference), calls intsect, aunion, and setdiff
6.22	dslash(x,y)	SETL x = x//y, calls boolex
6.23	abs(x)	SETL abs x (absolute value)
6.24	booland(x,y)	SETL x = x and y
	boolor(x,y)	SETL x = x or y
	boolex(x,y)	SETL x = x exor y
	boolimp(x,y)	SETL x = x implies y
	boolnot(x)	SETL x = not x
6.25	setltype(x)	SETL type x, calls type
	pair(x)	SETL pair x
6.26	newat(r)	SETL r = newat

6.27	min(x,y,r)	SETL r = x min y
	max(x,y,r)	SETL r = x max y
6.28	bot(x)	SETL bot x
	top(x)	SETL top x, calls pminus and bot
6.29	substr(s,i,l)	SETL s(i:l), retrieval, calls ncompallo
	ssubstr(s,i,l,r)	SETL s(i:l) = r (storage substring), calls substr and plus
6.30	dec(x)	SETL dec x
	oct(x)	SETL oct x
6.31	bitr(x)	SETL bitr x (numeric to/from boolean string conversion), calls bot
6.32	le(x,y)	SETL x le y, calls nextmem and elmtset
	lt(x,y)	SETL x lt y, calls le
6.33	pow(s)	SETL pow(s), calls nexpow and augaok
	npow(n,s)	SETL npow(n,s), calls nexnpow and augaok
	nexpow(c,s)	Computes 'next' subset of s, calls nexnpow
	nexnpow(c,n,s)	Computes 'next' subset of s of size n, calls nextmem and augaok
6.34	random(x)	SETL random x, main routine, calls ranint, ranreal, ranbool, ranchar, rantupl, and ranset
	ranint(n)	SETL random n, n an integer, calls ranbase
	ranreal(r)	SETL random r, r a real, calls ranbase
	ranbool(b)	SETL random b, b a boolean string. Calls ofbstr and ranint
	ranchar(c)	SETL random c, c a character string. Calls ofcstr and ranint
	rantupl(t)	SETL random t, t a tuple, calls oftuple and ranint
	ranset(s)	SETL random s, s a set, calls ranbase and nextmem
	ranbase(n)	Generates a non-SETL integer uniform on [0,n-1]
6.35	exp(x,y)	SETL x exp y, main routine, calls expii
	expii(x,y)	SETL x exp y, integers
6.36	tupaddl(t,x)	SETL t(#t+1) = x, calls space and ncompallo

6. DETAILED SPECIFICATIONS OF THE SETL PRIMITIVES

6.1 INITIALIZATION DATA AND UTILITY ROUTINES

6.1.1 TYPE CODES

```
INT = 0;        /* INTEGER. */
REAL = 1;       /* REAL (FLOATING POINT). */
BOOL = 2;       /* BOOLEAN STRING. */
CHAR = 3;       /* CHARACTER STRING. */
BLANK = 4;      /* BLANK ATOM (RETURNED BY NEWAT. ROUTINE). */
LABEL = 5;      /* LABEL VARIABLE. */
SUB = 6;        /* SUBROUTINE VARIABLE. */
FUN = 7;        /* FUNCTION VARIABLE. */
UNDEF = 8;      /* TYPE CODE USED FOR OMEGA. */
TUPLE = 9;      /* TUPLE. */
SET = 10;       /* SET. */
SPECPAIR = 11; /* TYPE CODE USED FOR THE SPECIAL PAIR (NOT A
                  SETL OBJECT). */
```

6.1.2 FIELDS OF OBJECTS

```
/* THESE FUNCTIONS MAY BE USED IN EITHER DEXTER OR SINISTER
   MODE. */

DEFINEF TYPE(OBJ); RETURN OBJ(1); END;
DEFINEF VALUE(OBJ); RETURN OBJ(2); END;
DEFINEF NEXTM(OBJ); RETURN OBJ(3); END;
DEFINEF NCOMPS(T); RETURN (VALUE(T))(1); END;
DEFINEF SPACE(T); RETURN (VALUE(T))(2); END;
DEFINEF TUP(T); RETURN (VALUE(T))(3); END;
DEFINEF NMEMBS(S); RETURN (VALUE(S))(1); END;
DEFINEF LOAD(S); RETURN (VALUE(S))(2); END;
DEFINEF HCODE(S); RETURN (VALUE(S))(3); END;
DEFINEF HTSIZE(S); RETURN (VALUE(S))(4); END;
DEFINEF HASHTABLE(S); RETURN (VALUE(S))(5); END;
```

6.1.3 HASHING PARAMETERS

```
HASHNLB = 66256;     /* FOR NULL BOOLEAN STRING. */
HASHNLC = 27182;     /* FOR NULL CHARACTER STRING. */
HASHNLT = 31416;     /* FOR NULL TUPLE. */
HASHNLS = 12345;     /* FOR NULL SET. */
HASHBLNK = 67447;    /* FOR BLANK ATOMS. */
HASHUNDF = 68421;    /* FOR UNDEFINED ATOM. */
```

6.1.4 TUPLE PARAMETERS

```
NCOMPSMALT = 2;  /* SMALLEST SPACE PARAMETER. */

DEFINEF NCOMPALLO(N);  /* INITIAL SPACE ALLOCATION FOR AN
                          N-TUPLE. */
RETURN NCOMPSMALT MAX. (((5*N)/4) + 1);
END NCOMPALLO;
```

6.1.5 SET PARAMETERS

```
MINHTSIZE = 2;       /* MINIMUM HASH TABLE SIZE. */
MAXHTSIZE = 32768;   /* MAXIMUM HASH TABLE SIZE. */

DEFINEF TOODENSE(S);  /* RETURNS ≠TRUE.≠ (THE META-OBJECT, NOT
                         THE SETL OBJECT) IF THE HASH TABLE OF
                         S IS TOO DENSELY USED. */
RETURN (LOAD(S) GT. 2*HTSIZE(S));
END TOODENSE;

DEFINEF TOOSPARSE(S);  /* RETURNS ≠TRUE.≠ IF THE HASH TABLE OF
                          S IS TOO SPARSELY USED. */
RETURN (2*LOAD(S) LT. HTSIZE(S));
END TOOSPARSE;
```

6.1.6 SEARCHING SETS

```
/* TO ACCURATELY REFLECT THE LITTLE CODE, EACH ENTRY IN A HASH
TABLE IS TREATED AS A LIST IN THIS SPECIFICATION.  THESE ~SET
LISTS~ ARE SCANNED USING A ~NEXT MEMBER~ FUNCTION NEXTM(M).
IN THIS SPECIFICATION, SET-LISTS ARE IMPLEMENTED BY ADDING A
THIRD COMPONENT TO EACH MEMBER, AS EXPLAINED IN SECTION 2.
NEXTM(M) IS SIMPLY M(3), AS SHOWN IN SECTION 5.1.2 ABOVE.  IN
THE LITTLE CODE NEXTM IS IMPLEMENTED WITH CHAIN POINTERS. */

DEFINEF INITM(OBJ, S);  /* GETS FIRST MEMBER OF S IN THE HASH
                           TABLE SLOT IMPLIED BY OBJ. */
RETURN (HASHTABLE(S))(HASH(OBJ)//HTSIZE(S) + 1);
END INITM;


/* BECAUSE SET SEARCHING IS SO UBIQUITOUS, THE FOLLOWING
MACROS ARE PROVIDED. */

BLOCK SEARCH(OBJ, S, M);
M = INITM(OBJ, S);   /* INITIALIZE M. */
(WHILE M NE. OM.  DOING M = NEXTM(M);)
END SEARCH;


BLOCK CONTSEARCH(M);
CONTINUE WHILE M NE. OM.
END CONTSEARCH;


BLOCK QUITSEARCH(M);
QUIT WHILE M NE. OM.
END QUITSEARCH;


BLOCK ENDSEARCH(M);
END WHILE M NE. OM.
END ENDSEARCH;
```

```
/* LOOPS THRCUGH SET-LISTS MAY BE CODED AS FOLLOWS:

                SEARCH(OBJECT, SET, M)
                ...
                CONTSEARCH(M);
                ...
                QUITSEARCH(M);
                ...
                ENDSEARCH(M);    */
```

6.1.7 SPECIAL SETL CONSTANTS

```
TRUE = <BOOL, 1B>;
FALSE = <BOOL, 0B>;
     /* NOTE:  THIS IMPLEMENTATION WILL USE THE SETL CONSTANT
     ≠TRUE.≠ FOR THE META-LANGUAGE CONSTANT, AND ≠TRUE≠
     (WITHOUT THE PERIOD) FOR THE OBJECT LANGUAGE CONSTANT
     (AND SIMILARLY FOR FALSE).  THE SPECIFICATION INCLUDES
     MANY EXPRESSIONS SUCH AS ≠IF (X EQ. TRUE)...≠, WHICH IS
     NOT EQUIVALENT TO ≠IF (X)...≠. */
NULLBSTR = <BOOL, NULB.>;      /* NULL BOOLEAN STRING. */
NULLCSTR = <CHAR, NULC.>;      /* NULL CHARACTER STRING. */
UNDEFWD = <UNDEF, 0>;          /* UNDEFINED ATOM (OMEGA). */
NULLTUPLE = <TUPLE, <0, 0, NULT.>>;  /* NULL TUPLE. */
NULLSET = <SET, <0, 0, HASHNLS, 0, NULT.>>;  /* NULL SET. */
```

6.1.8 MISCELLANEOUS VARIABLES

```
BLANKATOM = <BLANK, 0>;   /* USED BY NEWAT. ROUTINE. */
RANDSEED = 30*0B + 1B;    /* USED BY RANDOM ROUTINE. */
MAXRNP1 = 2 EXP. 31;      /* MAXIMUM RANDOM NUMBER + 1 (SEE
                             ROUTINE ≠RANBASE≠). */
STACK = NULT.;            /* USED BY STACKING ROUTINES
                             (PUSH, POP). */
MAXLINESZ = 130;          /* NUMBER OF CHARACTERS THAT WILL FIT
                             ON ONE LINE (ASSUMED TO BE THE
                             SAME FOR ALL FILES). */
```

6.1.9 STACKING ROUTINES

```
DEFINE PUSH(X);
STACK(↓STACK ← 1) = X;
RETURN;
END PUSH;


DEFINE POP(X);
X = STACK(↓STACK);
STACK(↓STACK) = 0M.;
RETURN;
END POP;

/* NOTE:  STACK IS INITIALIZED IN SECTION 5.1.8. */
```

6.2 INPUT/OUTPUT

6.2.1 EXTERNAL REPRESENTATION OF DATA

THE FOLLOWING DATA TYPES MAY BE READ IN WITH A READ STATEMENT:

INTEGERS	CHARACTER STRINGS
REALS	TUPLES
BOOLEAN STRINGS	SETS

IN ADDITION, THE DATA TYPES OF BLANK ATOMS, LABELS, SUBROUTINES, AND FUNCTIONS MAY BE PRINTED.

BELOW IS A BRIEF DESCRIPTION OF THE CHARACTER STRING, OR EXTERNAL, REPRESENTATION OF EACH DATA TYPE. THIS DESCRIPTION APPLIES BOTH TO INPUT AND OUTPUT OPERATIONS. HOWEVER, THERE IS GREATER FLEXIBILITY IN FORMATTING STRINGS FOR INPUT. THESE ALTERNATE FORMS ARE DESCRIBED BELOW FOR EACH CASE.

ON BOTH INPUT AND OUTPUT, SUCCESSIVE ITEMS ARE SEPARATED BY BLANKS. THE SLASH (/) IS AN OPTIONAL SEPARATOR THAT MAY BE USED TO MARK THE END OF THE EXTERNAL REPRESENTATION OF A COMPLICATED TUPLE OR SET. IT IS USED ON INPUT TO CHECK FOR POSSIBLE MISPARENTHETIZATION OR ILL-FORMED DATA. THE USUAL DELIMITERS ARE USED: < AND > FOR TUPLES, AND ≤ AND ≥ FOR SETS. THE INPUT STREAM MAY NOT CONTAIN COMMENTS. END-OF-FILE IS INDICATED BY THE DOLLAR SIGN CHARACTER ($).

THE EXTERNAL REPRESENTATION OF THE VARIOUS DATA TYPES IS:

1. INTEGERS: DECIMAL, OPTIONALLY PRECEDED BY A MINUS SIGN. NO EMBEDDED BLANKS OR COMMAS ARE ALLOWED.

2. REALS: NOT IMPLEMENTED YET.

3. BOOLEAN STRINGS: IN BINARY OR OCTAL FORM, OR IN A MIXTURE OF BOTH. THE BINARY PORTION PRECEDES THE OCTAL PORTION, AND IS SEPARATED FROM IT BY THE LETTER B. IF THE EXTERNAL FORM IS PURELY OCTAL, THE LETTER O IS APPENDED TO THE STRING. ON OUTPUT, THE LONGEST POSSIBLE OCTAL PORTION IS USED, WITH ONLY (N MOD. 3) BITS IN THE BINARY PORTION. EXAMPLES: 1B, 1B7, 1011001011B, 770, NULB. .

4. CHARACTER STRINGS: ENCLOSED IN SINGLE QUOTE MARKS (APOSTROPHES), WITH EMBEDDED QUOTES REPRESENTED BY TWO SUCCESSIVE QUOTES. ON INPUT, THE ENCLOSING

QUOTES MAY BE OMITTED IF THE STRING BEGINS WITH A LETTER AND DOES NOT CONTAIN ANY DELIMITERS, SUCH AS A BLANK. EXAMPLES: ≠ABC≠, ABC, NULC., ≠≠, ≠≠≠≠.

5. BLANK ATOMS: (OUTPUT ONLY). REPRESENTED BY A STRING OF THE FORM BLKN, WHERE N IS AN INTEGER IDENTIFYING THE ≠SERIAL NUMBER≠ OF THE BLANK ATOM. EXAMPLES: BLK1, BLK987.

6. LABELS, SUBROUTINES, AND FUNCTIONS: (OUTPUT ONLY), THE IDENTIFIER LAB, FUN, OR SUB, FOLLOWED BY A PERIOD, FOLLOWED BY THE SUBROUTINE≠S NAME (OBTAINED FROM A SYMBOL TABLE). EXAMPLES: LAB.LOOP, FUN.F.

7. TUPLES: IF E1, E2, ..., EN ARE THE EXTERNAL REPRESENTATIONS OF THE COMPONENTS OF THE TUPLE, THEN THE EXTERNAL REPRESENTATION OF THE TUPLE IS <E1,E2,...,EN>. THE NULL TUPLE IS WRITTEN <> ON OUTPUT, AND EITHER <> OR NULT. ON INPUT.

8. SETS: IF E1, E2, ..., EN ARE THE EXTERNAL REPRESENTATIONS OF THE MEMBERS OF THE SET, TAKEN IN AN ARBITRARY ORDER, THEN THE EXTERNAL FORM OF THE SET IS ≤E1,E2,...,EN≥. THE NULL SET IS WRITTEN ≤≥ ON OUTPUT, AND MAY BE WRITTEN ≤≥ OR NL. ON INPUT.

9. UNDEFINED ATOM: OM. (EITHER INPUT OR OUTPUT).

THERE ARE ABBREVIATING CONVENTIONS FOR THE OUTPUT FORM OF DEEPLY NESTED TUPLES AND SETS. THESE ARE DESCRIBED IN THE SPECIFICATION OF THE OUTPUT ROUTINES, BELOW.

THE TABLE BELOW SUMMARIZES THE SPECIAL VALUES THAT MAY BE WRITTEN IN PERIOD-DELIMITED FORM.

VALUE	PERIOD FORM	ALTERNATE FORM
TRUE	T.	1B
FALSE	F.	0B
NULL BOOL.	NULB.	(NONE)
NULL CHAR.	NULC.	≠≠
NULL TUPLE	NULT.	<>
NULL SET	NL.	≤≥
UNDEFINED	OM.	(NONE)

6.2.2 INPUT ROUTINES

THE INPUT ROUTINES FORM THE FOLLOWING HIERARCHICAL ORGANIZATION:

1. A LEXICAL SCANNER, ≠TOKREAD≠, WHICH EXTRACTS SUCCESSIVE TOKENS FROM THE INPUT STREAM, AND PRODUCES SETL CHARACTER STRINGS FOR EACH. THESE STRINGS CONTAIN EITHER DELIMITERS OR THE EXTERNAL FORM OF ATOMS. ≠TOKREAD≠ CALLS:

2. A SET OF CONVERSION ROUTINES FOR ATOMS: INTCHAR, BSTCHAR, AND RELCHAR, WHICH PRODUCE SETL INTEGERS, BOOLEAN STRINGS, AND REALS, RESPECTIVELY, FROM THE CORRESPONDING CHARACTER STRINGS. CHARACTER STRINGS THEMSELVES ARE CONVERTED (STRIPPED OF ENCLOSING QUOTES, AND WITH EMBEDDED DOUBLE QUOTES REPLACED WITH SINGLE ONES) BY TOKREAD ITSELF.

3. A PARSING ROUTINE ≠STLREAD≠, WHICH ASSEMBLES THE TOKENS INTO COMPOSITE OBJECTS (TUPLES AND SETS).

THE LEXICAL SCANNER

THE LEXICAL SCANNER CONSISTS OF THE MAIN ROUTINE ≠TOKREAD≠ AND THREE AUXILIARY SUBROUTINES CHRNEXT, PUTCHAR, AND FULLTOK.

THE LEXICAL SCANNER IS DESIGNED FROM THE IDEA OF A TABLE-DRIVEN ATOMATON. FOR EACH CHARACTER, A NUMERICAL ATTRIBUTE (CHARTYP) IS OBTAINED, AND THIS IS USED TO DRIVE A TRANSITION TABLE. HOWEVER, FOR EFFICIENCY IN THE ACTUAL RUN TIME LIBRARY ROUTINE, WE DON≠T ACTUALLY INTERPRET A TABLE AT EXECUTION TIME. INSTEAD, THE ROUTINE CONSISTS OF A SERIES OF IF AND GO TO STATEMENTS WHOSE STRUCTURE MIRRORS THAT OF THE TRANSITION TABLE.

THE CHARACTER TYPES AND THEIR NUMERICAL ATTRIBUTES ARE:

CHARACTER	#CHARTYP#
0	1
B	2
LETTER OTHER THAN O, B	3
INTEGER	4
QUOTE	5
BLANK	6
PERIOD	7
DELIMITER: COMMA,<,>,≤,≥	8
OTHER SPECIAL CHARACTERS	9

THE STATES OF THE ATOMATON ARE:

1. STARTTOK: BEGINNING OF A TOKEN (NO CHARACTER READ).
2. BITNUM: SCANNING AN INTEGER, REAL, OR BOOLEAN STRING.
3. BITEXACT: SCANNING A BOOLEAN STRING IN OCTAL FORM.
4. BITTRAIL: SCANNING A BOOLEAN STRING IN BINARY OR MIXED FORM.
5. REALNUM: SCANNING A REAL NUMBER.
6. CHRBLANK: SCANNING A CHARACTER STRING DELIMITED BY BLANKS.
7. CHRQUOT: SCANNING A CHARACTER STRING DELIMITED BY QUOTES.
8. QUOTQUOT: CHARACTER IS A QUOTE WITHIN A QUOTED CHARACTER STRING.
9. ERRORD: UNRECOGNIZABLE STRING.

THE TRANSITION TABLE TAKES THE FOLLOWING FORM:

	0	B	ALPHA	INT.	QUOTE	BLANK	PER.	DELIM	OTHE
1.STARTTOK	CFLAG GO 6	CFLAG GO 6	CFLAG GO 6	GO 2	CFLAG GO 7	GO 1	GO 9	RETDELIM	GO 9
2.BITNUM	GO 3	GO 4	GO 9	GO 2	GO 9	RETINT	GO 5	BACKONE RETINT	GO 9
3.BITEXACT	GO 9	GO 9	GO 9	GO 9	GO 9	RETBIT	GO 9	BACKONE RETBIT	GO 9
4.BITTRAIL	GO 9	GO 9	GO 9	GO 4	GO 9	RETBIT	GO 9	BACKONE RETBIT	GO 9
5.REALNUM	GO 9	GO 9	GO 9	GO 5	GO 9	RETREAL	GO 9	BACKONE RETREAL	GO 9
6.CHRBLANK	GO 6	GO 6	GO 6	GO 6	GO 9	RETCHAR	GO 6	BACKONE RETCHAR	GO 6
7.CHARQUOT	GO 7	GO 7	GO 7	GO 7	GO 8	GO 7	GO 7	GO 7	GO 7
8.QUOTQUOT	GO 9	GO 9	GO 9	GO 9	GO 7	RETCHAR	GO 9	BACKONE RETCHAR	GO 9
9.ERRORD									

THE ACTIONS LISTED IN THE ABOVE TABLE ARE:

CFLAG: SET FLAG ≠READFLAG≠ TO INDICATE THAT A CHARACTER STRING IS BEING SCANNED.

RETDELIM: THE TOKEN IS A DELIMITER. RETURN IT.

RETINT: THE TOKEN IS AN INTEGER. CALL THE CONVERSION ROUTINE ≠INTCHAR≠ AND RETURN.

RETBIT: THE TOKEN IS A BOOLEAN STRING. CALL ≠BSTCHAR≠ AND RETURN.

RETREAL: THE TOKEN IS A REAL NUMBER. CALL ≠RELCHAR≠ AND RETURN.

BACKONE: THE LAST CHARACTER READ IS THE BEGINNING OF A NEW TOKEN. RETURN IT TO THE INPUT STRING, AND BACKSPACE THE CHARACTER POINTER BY ONE.

ERRORD: THE STRING DOES NOT CORRESPOND TO A VALID DATA TYPE. PRINT IT, TOGETHER WITH THE INPUT RECORD FROM WHICH IT WAS READ, AND ABORT.

```
/*                          STLREAD                                  */

      DEFINEF STLREAD(FILE, L);

      /* THE PARSING ROUTINE ≠STLREAD≠ OBTAINS SUCCESSIVE TOKENS
      FROM ≠TOKREAD≠ AND ASSEMBLES THEM INTO SETS AN TUPLES.  IT
      IS DRIVEN BY A NUMERICAL ATTRIBUTE ≠TOKIND≠ OBTAINED FOR EVERY
      TOKEN, FROM THE TABLE ≠SETLDEL≠.  TOKENS ARE EITHER ATOMS (FOR
      WHICH TOKIND = 8) OR DELIMITERS.
           A STACK ≠PARTSTACK≠ IS USED TO STORE PARTLY BUILT
      COMPOSITE OBJECTS.  THE VARIABLE ≠CURRDEL≠ CONTAINS THE
      ≠TOKIND≠ CORRESPONDING TO THE LAST OPENING DELIMITER
      ENCOUNTERED.  WHEN A CLOSING DELIMITER IS FOUND, A MATCHING
      OPENER HAS TO APPEAR IN CURRDEL.  AT THIS POINT THE COMPOSITE
      ITEM BEING SCANNED (THE CURRENT TOP OF PARTSTACK) IS COMPLETE.
      THIS COMPLETED ITEM IS EITHER RETURNED FROM ≠STLREAD≠ (IF IT
      IS THE ONLY ITEM PRESENT IN PARTSTACK) OR IT IS ADDED TO THE
      NEXT LOWER ELEMENT ON PARTSTACK, WHICH BECOMES THE CURRENT
      COMPOSITE ITEM BEING BUILT. */

      /* SET UP A CONSTANT TABLE FOR DELIMITER TYPE NUMBERS. */
      SETLDEL= <TUPLE, <CHAR,≠≤≠>,<CHAR,≠<≠>,<CHAR,≠≥≠>,<CHAR,≠>≠>,
                       <CHAR,≠,≠>,<CHAR,≠/≠>,<CHAR,≠$≠>>;

      PARTSTACK = NULLTUPLE;
START: TOK = TOKREAD(FILE, L);
      (1 <= ~K <= NCOMPS(SETLDEL))
        IF TOK EQ. SETLDEL(K) THEN QUIT ~K;;;

      /* ON FALL-THROUGH, K IS EQUAL TO 8. */

      TOKIND = K;

      FLOW
                                SPECIAL $
          STARTNEW  $                          STACKEMPTY $
      STARTIT,      TERMIN $                RETURNIT,  ENTERIT,
                MATCHES $                      COMMA $
          FIRST $     ERROR,                STACKEMPTY $      SLASH $
      RETFIRST, ENDIT,                   ERROR,  TO START,
                                                 STACKEMPTY $     CRASH,
                                           TO START,       ERRPRINT;

  SPECIAL:=  TOKIND LT. 8;

  STACKEMPTY:=PARTSTACK EQ. NULLTUPLE;

  RETURNIT:  RETURN TOK;
```

```
ENTERIT:    IF CURRDEL EQ.1 THEN ALGMENT(TOK,TOPP(PARTSTACK));
                            ELSE TUPADD1(TOPP(PARTSTACK),TOK);;

STARTNEW:= TOKIND LE. 2;

TERMIN:=   TOKIND LE. 4;

MATCHES:=  CURRDEL EQ. (TOKIND-2);

STARTIT:   TUPADD1(PARTSTACK,IF TOKIND EQ.1 THEN NULLSET
                                            ELSE NULLTUPLE);
           CURRDEL=TOKIND;

FIRST:=    NCOMPS(PARTSTACK) EQ. 1;

RETFIRST:  RETURN (TUP(PARTSTACK))(1);

ENDIT:     COMPOSITE=TOPP(PARTSTACK);TOPP(PARTSTACK)=UNDEFWD;
           IF TYPE(TOPP(PARTSTACK)) EQ. SET THEN
             AUGMENT(TOK, TOPP(PARTSTACK));  CURRDEL = 1;
           ELSE
             TUPADD1(TOPP(PARTSTACK), TOK);  CURRDEL = 2;;

COMMA:=    TOK EQ.<CHAR,#,#>;

SLASH:=    TOK EQ.<CHAR,#/#>;

CRASH;     PRINT. #END OF FILE ON ATTEMPT TO READ#; CRASH;

ERROR:     X = TOKREAD(FILE, L);   /* SKIP TO $ OR /. */
           (WHILE X NE. <CHAR, #$#>  AND.  X NE. <CHAR. #/#>)
             X = TOKREAD(FILE, L);;

ERRPRINT:  PRINT. #ILLEGAL CONFIGURATION DETECTED ON ATTEMPTED #
                * #FILE READ.#;
           RETURN UNDEFWD;

     END FLOW;

     GO TO START;

     DEFINEF TOPP(X); RETURN (TUP(X))(NCOMPS(X)); END TOPP;
     END STLREAD;
```

6.2.3 OUTPUT ROUTINES

SETL OBJECTS ARE WRITTEN ON FILES ACCORDING TO THE FORMAT DESCRIBED IN SECTION 5.2.1. HOWEVER, A SPECIAL FORMAT IS USED TO PRINT DEEPLY NESTED SETS AND TUPLES, TO IMPROVE READABILITY. IF THE OBJECT IS NESTED MORE DEEPLY THAN SOME USER-DEFINED PARAMETER (MAXDEP), THEN THOSE ITEMS IN THE OBJECT WHICH ARE AT A GREATER DEPTH ARE ABBREVIATED. THE ABBREVIATION CONSISTS OF AN ASTERISK FOLLOWED BY A DECIMAL NUMBER, IN THE EXTERNAL REPRESENTATION. THIS IS PRINTED IN PLACE OF THE OBJECT IN THE ORIGINAL TUPLE OR SET, AND SUBSEQUENTLY THE ABBREVIATION LABEL AND THE OBJECT ITSELF ARE PRINTED. ABBREVIATION ALSO TAKES PLACE IF THE EXTERNAL FORM OF A TUPLE OR SET IS VERY LONG (MORE THAN A CERTAIN NUMBER OF LINES).

FOR EXAMPLE, IF MAXDEP = 2, THE SET

{A,B,<C,{D,E},F>,G}

WOULD BE PRINTED AS:

```
{A,B,<C,*1,F>,G}
  *1 {D,E}
```

IF MAXDEP IS ZERO, NO ABBREVIATING TAKES PLACE.

AT PRESENT THE INPUT ROUTINES ONLY ACCEPT UNABBREVIATED REPRESENTATIONS. THEREFORE, WHEN WRITING ON A FILE THAT IS TO BE SUBSEQUENTLY READ IN, MAXDEP MUST BE SET TO ZERO.

A SET OF ATOM CONVERSION ROUTINES (NOT SPECIFIED HERE) TRANSFORMS INTEGERS, REALS, BOOLEAN STRINGS, CHARACTER STRINGS, ETC., INTO THEIR RESPECTIVE EXTERNAL FORMS.

```
/*                    PRINTER                                    */

DEFINE PRINTER(OBJ,FILE,MAXDEP);

/* THIS IS THE MAIN ROUTINE FOR THE OUTPUT PACKAGE.  IT
INITIALIZES PARAMETERS FOR THE RECURSIVE ROUTINES #PRINTC#
AND #CHAROUT#.

  OBJ IS THE SETL OBJECT TO BE WRITTEN.
  FILE IS THE FILE ON WHICH TO WRITE IT.
  MAXDEP IS THE MAXIMUM DEPTH OF NESTING, ABOVE WHICH ITEMS
     ARE ABBREVIATED. */

DEPNOW = 0;          /* DEPTH OF NESTING REACHED WHILE SCANNING
                     AN ITEM. */
DEPTH = 0;           /* DEPTH OF ABBREVIATION. */
POSITN = 0;          /* CHARACTER POSITION FOR THE BEGINNING
                     OF THE NEXT LINE.  SUCCESSIVE LEVELS OF
                     ABBREVIATIONS ARE INDENTED 2*DEPTH
                     SPACES. */
LINES = 3;           /* MAXIMUM NUMBER OF LINES, ABOVE WHICH
                     THE FOLLOWING ITEMS ARE ABBREVIATED. */
LINENO = 0;          /* NUMBER OF LINES USED SO FAR FOR AN
                     ITEM. */
LINNOW = 0;          /* NUMBER OF LINES USED BEFORE BEGINNING
                     TO SCAN AN ABBREVIATED ITEM. */

PRINTC(OBJ,NULLCSTR,FILE,MAXDEP);
RETURN;
END PRINTER;
```

```
/*                          PRINTC                                  */

DEFINE PRINTC(OBJ,ALABEL,FILE,MAXDEP);

/* THE SECOND ARGUMENT OF PRINTC IS THE LABEL THAT INDICATES
THE LOCATION FROM WHICH OBJ WAS ABBREVIATED. ON FIRST ENTRY TO
≠PRINTC≠ THAT LABEL IS THE NULL STRING.  AS AN OBJECT IS
BEING SCANNED FOR PRINTING, A TUPLE CONTAINING SUCCESSIVE
ABBREVIATED ITEMS IS BUILT.  THE COMPONENTS OF THIS TUPLE,
≠TABBR≠, ARE SUBSEQUENT INPUT PARAMETERS FOR PRINTC.  THIS
ROUTINE IS THEREFORE FULLY RECURSIVE. */

TABBR = NULLTUPLE;  /* INITIALIZE. */
DEPNOW = 0;
MAXD = MAXDEP;      /* SAVE PARAMETER THAT MAY BE MODIFIED IN
                       CHAROUT. */

POST(FILE, ALABEL); /* POST THE LABEL (MAY BE NULL). */
POST(FILE, <CHAR, ≠ ≠>);
LINNOW = LINENO;
CHAROUT(OBJ, FILE, MAXDEP);   /* POST THE ITEM ITSELF.  IN THE
                                 PROCESS, TABBR WILL BE BUILT. */
MAXDEP = MAXD;      /* RESTORE PARAMETER. */
LINES = 3;          /* RESTORE. */

DEPTH = DEPTH + 1;
POSITN = (2*DEPTH)//(MAXLINESZ/4); /* EACH SUCCESSIVE LEVEL
                       OF ABBREVIATION IS INDENTED 2 SPACES, UP
                       TO A MAXIMUM OF 1/4 OF THE PAGE SIZE. */

(1 ≤ ~K ≤ NCOMPS(TABBR))

  /* BUILD THE LABEL CORRESPONDING TO EACH COMP. OF TABBR. */
  LABELK = <CHAR, ≠*≠ + VALUE(LABEL) + DEC.K>;

  /* PRINT EACH COMPONENT OF TABBR. */
  PRINTC((TUP(TABBR))(K), LABELK, FILE, MAXDEP);
  END ~1;

DEPTH = DEPTH - 1;
POSITN = 0;
RETURN;
END PRINTC;
```

```
DEFINE CHAROUT(OBJ,ALABEL,FILE,MAXDEP);

/* CHAROUT CALLS THE ATOM CONVERSION ROUTINES, AND HANDLES THE
RECURSION FOR THE CHARACTER CONVERSION OF TUPLES AND SETS.  IT
ALSO CHECKS FOR THE NEED OF ABBREVIATING, AND ADDS ITEMS TO
≠TABBR≠ WHENEVER REQUIRED. */

IF TYPE(OBJ) LT. TUPLE THEN  /* ATOM. */
    IF TYPE(OBJ) EQ. INT   THEN POST(FILE,CHARINT(OBJ));;
    IF TYPE(OBJ) EQ. REAL  THEN POST(FILE,CHARREL(OBJ));;
    IF TYPE(OBJ) EQ. BOOL  THEN POST(FILE,CHARBST(OBJ));;
    IF TYPE(OBJ) EQ. CHAR  THEN POST(FILE,CHARCST(OBJ));;
    IF TYPE(OBJ) EQ. BLANK THEN POST(FILE,CHARBLK(OBJ));;
    IF TYPE(OBJ) EQ. LABEL THEN POST(FILE,CHARLBL(OBJ));;
    IF TYPE(OBJ) EQ. SUB   THEN POST(FILE,CHARSUB(OBJ));;
    IF TYPE(OBJ) EQ. FUN   THEN POST(FILE,CHARFUN(OBJ));;
    IF TYPE(OBJ) EQ. UNDEF THEN POST(FILE,CHARUND());;
ELSE

  /* THE OBJECT IS COMPOSITE.  CHECK FOR THE NEED OF ABBREV.*/

  IF MAXDEP EQ. 0 THEN GO TO SKIP;;

  /* THE NULL TUPLE AND SET ARE NEVER ABBREVIATED. */
  IF (OBJ EQ.NULLTUPLE) OR. (OBJ EQ. NULLSET) GO TO SKIPABBR;;

  M = LINENO - LINNOW;   /* NUMBER OF LINES THAT THE PRESENT
                            ITEM HAS FILLED SO FAR. */
  IF(DEPNOW GE.MAXDEP) OR. (M.GE.LINES) THEN /*ABBREVIATE.*/
    /* IF OBJ IS A LONG TUPLE AND WE ARE SCANNING A SET, IT IS
    NECESSARY TO COPY OBJ, BECAUSE NEXTMEM DESTROYS  ITS FIRST
    ARGUMENT.*/
    IF ((FLAGSET EQ. TRUE) AND. (TYPE(OBJ) EQ. TUPLE) AND.
                               (NCOMPS(OBJ) GT.3)) THEN
      OBJ = CONCATT(NULLTUPLE, OBJ);;

    /*PUT THE OBJECT IN THE TUPLE OF ABBREVIATIONS AND POST
    ITS LABEL INSTEAD.*/

    TUPADD1(TABBR,OBJ);
    POST(FILE, ALABEL);
    POST(FILE, <CHAR, ≠*≠>);
    RETURN;
    END IF (DEPNOW GE. MAXDEP);
```

```
        /*CHECK DENSITY OF ABBREVIATIONS,AND INCREASE MAX. ALLOWABLE
        VALUES OF PARAMETERS IF NECESSARY.*/

SKIP:   IF M GT. 0 AND. ((NCOMPS(TABER)/M) GT. 6) THEN
          MAXDEP = MAXDEP + 4;
          LINES = LINES + 2;;

        DEPNOW = DEPNOW + 1;    /* BEGIN SCAN OF A COMPOSITE OBJECT*/
        IF TYPE(OBJ) EQ. TUPLE THEN
          FLAGSET=FALSE;
          (1 ≤ ^K ≤ NCOMPS(OBJ))
            IF K EQ. 1 THEN POST(FILE, <CHAR, ≠<≠>)
                       ELSE POST(FILE, <CHAR, ≠,≠>);;

            CHAROUT(TUP(OBJ)(K),ALABEL,FILE,MAXDEP);
            END ^K;

          POST(FILE, <CHAR, ≠>≠>);

        ELSE
          /* OBJECT IS A SET. */
          FLAGSET = TRUE;
          C = 0;
LOOP:       IF C EQ. 0 THEN POST(FILE, <CHAR, ≠≤≠>)
                       ELSE POST(FILE, <CHAR, ≠,≠>);;
            M = NEXTMEM(C, OBJ);
            IF M EQ. UNDEFWD THEN GO TO OUT;;
            CHAROUT(M, ALABEL, FILE, MAXDEP);
            GO TO LOOP;

OUT;        POST(FILE, <CHAR, ≠≥≠>);
            END IF TYPE(OBJ) EQ. TUPLE;

        DEPNOW = DEPNOW - 1;
        END IF TYPE(OBJ) LT. TUPLE;

      RETURN;
      END CHAROUT;
```

```
/*                          POST                                    */

SUBR POST(FILE,X);

/* THIS SUBRCUTINE PLACES THE CHARACTER STRING X ON THE LINE
TO BE PRINTED, AND PASSES COMPLETED LINES TO THE LOWER-LEVEL
ROUTINE #OUTSTR# FOR PRINTING.
     THE VALUE X = 0 (NOT A CHARACTER STRING) IS USED AS A
SIGNAL TO START A NEW LINE.
     THE VARIABLE #POSITN#, WHICH IS SET IN PRINTER AND
PRINTC, IS AN INPUT TO POST.  WHENEVER A NEW LINE IS BEGUN IT
IS INITIALIZED TO #POSITN# BLANKS. */

IF X EQ. 0 THEN
  OUTSTR(FILE,LINE);  /*PRINT PRESENT BUFFER.*/
  LINE = <CHAR, POSITN*# #>;;
COPYLT = 0;

(WHILE COPYLT LT. ↓VALUE(X) DOING COPYLT=COPYLT+LPIECE;)
  IF ↓VALUE(X) EQ. MAXLINESZ THEN
    OUTSTR(FILE,LINE);
    LINENO = LINENO + 1;
    LINE = <CHAR, POSITN*# #>;;
  LPIECE= (↓VALUE(X)-COPYLT) MIN.(MAXLINESZ-↓VALUE(LINE));
  IF LPIECE EQ.↓VALUE(X) THEN PIECEX = X;
  ELSE PIECEX = X(COPYLT+1:LPIECE);;
  VALUE(LINE) = VALUE(LINE) + VALUE(PIECEX);
  END WHILE COPYLT;
RETURN;
END POST;
```

6.3 ERROR ROUTINES

```
DEFINE ERRIMP(MESSAGE, OBJECT);
PRINT. ≠*-***** IMPLEMENTATION ERROR. UNUSED TYPE CODE DETECTED≠
     + ≠ BY ≠ + MESSAGE, OBJECT;
EXIT;
END ERRIMP;


DEFINE ERRTYPE(MESSAGE, OBJECT);
PRINT. ≠*-***** INVALID TYPE CODE FOR ROUTINE ≠ + MESSAGE,
     OBJECT;
EXIT;
END ERRTYPE;


DEFINE ERRVAL(MESSAGE, OBJECT);
PRINT. ≠*-***** INVALID DATA VALUE FOR ROUTINE ≠ + MESSAGE,
     OBJECT;
EXIT;
END ERRVAL;


DEFINE ERRMSG(MESSAGE);
PRINT. ≠*-***** ≠ + MESSAGE;
EXIT;
END ERRMSG;


DEFINE ERRIMPL(MESSAGE);
PRINT. ≠*-***** IMPLEMENTATION ERROR. ≠ + MESSAGE;
EXIT;
END ERRIMPL;


DEFINE ERRMIX(MESSAGE, OBJ1, OBJ2);
PRINT. ≠*-***** INVALID (MIXED) DATA TYPES FOR ROUTINE ≠
     + MESSAGE, OBJ1, OBJ2;
EXIT;
END ERRMIX;
```

6.4 HASHING ROUTINE

```
DEFINEF HASH(OBJECT);

/* THIS ROUTINE CALCULATES AND RETURNS THE HASH CODE FOR THE
GIVEN SETL OBJECT.  THE HASH CODE RETURNED BY THIS ROUTINE IS
AN INTEGER FROM ZERO TO THE MAXIMUM HASH TABLE SIZE MINUS ONE.
IT IS ALWAYS USED MODULO THE SIZE OF THE HASH TABLE INVOLVED
(SEE INITM, SECTION 5.1.6).
     THE HASH CODE OF A TUPLE IS THE HASH CODE OF ITS FIRST
COMPONENT.  THE HASH CODE OF A SET IS MAINTAINED WITH THE SET
(AND THIS ROUTINE MERELY RETRIEVES IT). */

X = OBJECT;

/* THE HASH CODE OF A TUPLE IS THE HASH CODE OF ITS FIRST
   COMPONENT. */

(WHILE (TYPE(X) EQ. TUPLE  AND.  NCOMPS(X) NE. 0)
        OR. TYPE(X) EQ. SPECPAIR)
  X = (VALUE(X)) (1);
  END;

IF TYPE(X) EQ. SET THEN RETURN HCODE(X);;
IF X EQ. UNDEFWD THEN RETURN HASHUNDF;;
IF X EQ. NULLTUPLE THEN RETURN HASHNLT;;

/* X IS NOW KNOWN TO BE AN ATOM.  FOR THIS ALGORITHM WE
ASSUME THE EXISTENCE OF AN ≠UNSPEC≠ ROUTINE WHICH, LIKE THE
PL/I UNSPEC, CONVERTS ANY ATOM INTO A BIT STRING THAT RE-
FLECTS THE OBJECT≠S INTERNAL REPRESENTATION.  FOR LABELS,
SUBROUTINES, AND FUNCTIONS, THE BIT STRING MAY BE THE
ADDRESS OF THE ROUTINE IF IT IS UNMOVEABLE, OR IT MAY BE
THE ROUTINE≠S CODE IF IT DOES NOT MODIFY ITSELF. */

BITS = UNSPEC(VALUE(X));

/* NOW FORM THE EXCLUSIVE OR OF ALL THE BITS OF X, TAKING
THEM 60 AT A TIME, FIRST INITIALIZE TO FORCE A LACK OF
CORRELATION BETWEEN THE INTEGER ZERO AND NULL OBJECTS, ETC. */

HASHX = 0B;
IF TYPE(X) EQ. BOOL  THEN HASHX = UNSPEC(HASHNLB);;
IF TYPE(X) EQ. CHAR  THEN HASHX = UNSPEC(HASHNLC);;
IF TYPE(X) EQ. BLANK THEN HASHX = UNSPEC(HASHBLNK);;

HASHX = HASHX // BITS(1:(↓BITS//60));  /*(BITS(1:0) = < >),*/
K = ↓BITS//60 * 1;
```

```
(WHILE K LT. 4BITS) HASHX = HASHX//BITS(K:60);  K = K+60; END;

/* AT THIS POINT HASHX IS A 60-BIT HASH CODE OF THE INPUT
ARGUMENT.  THIS IS NEXT REDUCED TO LOG2(MAXHTSIZE) BITS BY
FIRST FORMING THE EXCLUSIVE OR OF THE LEFT AND RIGHT HALVES,
AND THEN DIVIDING THE 30-BIT RESULT BY MAXHTSIZE-1.  THE
REMAINDER IS THE FINAL HASH CODE RETURNED BY THIS ROUTINE.
     THE REASON FOR FIRST REDUCING TO 30 BITS IS THAT DIVISION
OF FULL WORD QUANTITIES IS AWKWARD ON THE CDC-6600.  THE
REASON FOR THE FINAL DIVISION IS TO CAUSE ALL BITS OF THE
ORIGINAL OBJECT TO CONTRIBUTE TO EACH BIT OF THE RESULT (IN
SOME OBSCURE WAY).
     FOR A DISCUSSION OF THE MERITS OF COMPUTING HASH CODES BY
REMAINDERING, SEE "KEY-TO-ADDRESS TRANSFORMATION TECHNIQUES, A
FUNDAMENTAL PERFORMANCE STUDY ON LARGE EXISTING FORMATTED
FILES", COMMUNICATIONS OF THE ACM, VOLUME 14 NUMBER 4
(APRIL 1971). */

HASHX = HASHX(1:30) // HASHX(31:60);
RETURN (HASHX AS. INT.) // (MAXHTSIZE-1);
END HASH;
```

6.5 ATOM ROUTINE

```
DEFINEF ATOM(A);  /* ENTRY POINT FOR SETL ATOM.A. */

RETURN <BOOL, (TYPE(A) LE. UNDEF)>;  /* NOTE THAT OMEGA IS
     CONSIDERED TO BE AN ATOM. */
END ATOM;
```

6.6 NUMBER OF ELEMENTS ROUTINE

```
DEFINEF NELT(A);  /* ENTRY POINT FOR SETL #A
                     (# = NUMBER SIGN). */

IF TYPE(A) EQ. SET THEN RETURN <INT, NMEMRS(A)>;;
IF TYPE(A) EQ. TUPLE THEN RETURN <INT, NCOMPS(A)>;;
IF TYPE(A) EQ. CHAR THEN RETURN <INT, #VALUE(A)>;;
IF TYPE(A) EQ. BOOL THEN RETURN <INT, #VALUE(A)>;;
ERRTYPE(≠NELT(A), A IS≠, A);
END NELT;
```

6.7 EQUALITY TEST

```
DEFINEF EQUAL(A, B);  /* ENTRY POINT FOR THE SETL A EQ. B. */

/*    TWO SETL OBJECTS ARE CONSIDERED EQUAL IF THEY HAVE THE
SAME TYPE AND THE SAME VALUE, OR BOTH OBJECTS ARE UNDEFINED
(OMEGA).  THUS, A REAL IS NEVER EQUAL TO AN INTEGER, A BOOLEAN
STRING IS NEVER EQUAL TO A CHARACTER STRING, A TUPLE IS NEVER
EQUAL TO A SET, ETC.  ANY TWO SETL OBJECTS MAY BE COMPARED,
AND THE RESULT IS ALWAYS EITHER TRUE OR FALSE.  IN PARTICULAR,
(OMEGA EQ. OMEGA) IS TRUE, AND (OMEGA EQ. ANYTHING ELSE) IS
FALSE.
     TO BE EQUAL, STRINGS, TUPLES, AND SETS MUST HAVE THE SAME
NUMBER OF ELEMENTS.  STRINGS AND TUPLES MUST HAVE CORRESPON-
DING ELEMENTS EQUAL.  FOR SETS, IT IS SUFFICIENT THAT EVERY
MEMBER OF EACH SET BE CONTAINED IN THE OTHER.
     THIS FIRST LEVEL ROUTINE IS NON-RECURSIVE.
     NOTE:  THERE IS NO ROUTINE FOR A NE. B.  INSTEAD, IT IS
ASSUMED THAT THE COMPILER WILL USE EQUAL, AND REVERSE ITS
TEST-AND-BRANCH LOGIC. */

/* MAKE A QUICK DETERMINATION WITHOUT INVOLVING CALLS ON
RECURSIVE ROUTINES, IF POSSIBLE. */

K = EQBASIC(A, B);  /* K = 1 IF A NE. B,
                           2 IF A EQ. B,
                           3 IF A AND B ARE NON-NULL TUPLES OF
                             EQUAL LENGTH, OR ARE SPECIAL
                             PAIRS,
                           4 IF A AND B ARE NON-NULL SETS OF
                             EQUAL SIZE AND HASH CODES. */

GO TO <NEQ, EQ, TUPLES, SETS>(K);
  NEQ:    RETURN FALSE;
  EQ:     RETURN TRUE;
  TUPLES: RETURN EQTUPNT(A, B);
  SETS:   RETURN SUBSETNT(A, B);

END EQUAL;
```

```
/*                          EQTUPNT                              */

DEFINEF EQTUPNT(A, B);

/* RECURSIVE ROUTINE FOR NON-TRIVIAL TUPLE COMPARISON.  A AND
B ARE KNOWN TO BE NON-NULL, EQUAL LENGTH TUPLES OR SPECIAL
PAIRS.  THE TUPLES ARE COMPARED WITHOUT MAKING A RECURSIVE
CALL UNLESS A COMPONENT IS A TUPLE OR A SET. */

IF TYPE(A) EQ. TUPLE THEN
  T1 = TUP(A);
  T2 = TUP(B);
ELSE /* SPECIAL PAIRS. */
  T1 = VALUE(A);
  T2 = VALUE(B);
  END IF;
I = 1;
(WHILE I LT. #T1)
  K = EQBASIC(T1(I), T2(I));
  GO TO <NEQ, EQ, TUPTUPLES, TUPSETS>(K);
    NEQ:        RETURN FALSE;
    EQ:         CONTINUE WHILE I;
    TUPTUPLES:  IF EQTUPNT(T1(I), T2(I)) EQ. TRUE THEN
                  CONTINUE WHILE I;
                ELSE RETURN FALSE;;
    TUPSETS:    IF SUBSETNT(T1(I), T2(I)) EQ. TRUE THEN
                  CONTINUE WHILE I;
                ELSE RETURN FALSE;;
  END WHILE I;

RETURN TRUE;  /* ALL COMPONENTS ARE EQUAL. */
END EQTUPNT;
```

```
/*                          SUBSETNT                                  */

DEFINEF SUBSETNT(A, B);

/* RECURSIVE ROUTINE FOR NON-TRIVIAL SET COMPARISON.  SETS ARE
KNOWN TO BE NON-NULL, OF EQUAL SIZE, AND TO HAVE EQUAL HASH
CODES.  FOR THIS CASE THE EQUALITY TEST IS EQUIVALENT TO A
SUBSET TEST, I.E., THE ROUTINE CHECKS TO SEE IF EACH MEMBER
OF A IS IN B.  IF SO, THE SETS MUST BE EQUAL.
     FOR EFFICIENCY REASONS, THE MEMBERSHIP TEST ROUTINE
(ELMTSETSMP) IS NOT USED TO DETERMINE WHETHER OR NOT EACH
OBJECT IN A IS A MEMBER OF B (TO DO SO WOULD MAKE ELMTSETSMP
A RECURSIVE ROUTINE, AS IT CALLS EQUAL).  INSTEAD, THE LOGIC
OF THE MEMBERSHIP TEST IS DUPLICATED HERE.
     THE SETS ARE COMPARED WITHOUT MAKING A RECURSIVE CALL
UNLESS A MEMBER IS A TUPLE OR A SET. */

/* SCAN THE SLOTS OF THE HASH TABLE OF A. */
(^SLOT → HASHTABLE(A))
  /* SCAN THE MEMBERS IN THE CURRENT SLOT OF HASHTABLE(A). */
  MA = SLOT;
  (WHILE MA NE. OM.  DOING MA = NEXTM(MA);)
    /* SEARCH SET B FOR MEMBER MA. */
    SEARCH(MA, B, MB)
      K = EQBASIC(MA, MB);
      GO TO <NEQ, EQ, SETTUPLES, SETSETS>(K);
        NEQ:        CONTSEARCH(MB);
        EQ:         CONTINUE WHILE MA NE. OM.;  /* GET NEXT
                                 MEMBER OF SET A TO TEST. */
        SETTUPLES:  IF EQTUPNT(MA, MB) EQ. TRUE THEN
                      CONTINUE WHILE MA NE. OM.;
                    ELSE CONTSEARCH(MB);;
        SETSETS:    IF SUBSETNT(MA, MB) EQ. TRUE THEN
                      CONTINUE WHILE MA NE. OM.;
                    ELSE CONTSEARCH(MB);;
      ENDSEARCH(MB);
    /* MEMBER MA WAS NOT FOUND IN SET B. */
    RETURN FALSE;
    END WHILE MA NE. OM.;
  END ^SLOT;
RETURN TRUE;
END SUBSETNT;
```

```
/*                          EQBASIC                              */

DEFINEF EQBASIC(A, B);

/* THIS NON-RECURSIVE ROUTINE COMPARES A AND B AND RETURNS
WITH AN INTEGER CODED AS FOLLOWS:
          1: OBJECTS ARE NOT EQUAL.
          2: OBJECTS ARE EQUAL.
          3: OBJECTS ARE NON-NULL TUPLES OF EQUAL LENGTH OR
             ARE SPECIAL PAIRS (THE COMPONENTS HAVE NOT BEEN
             EXAMINED).
          4: OBJECTS ARE NON-NULL SETS OF EQUAL SIZE AND
             EQUAL HASH CODES (THE MEMBERS HAVE NOT BEEN
             EXAMINED).  */

IF TYPE(A) NE. TYPE(B) THEN RETURN 1;;
IF ATOM(A) EQ. TRUE THEN
  IF VALUE(A) EQ. VALUE(B) THEN RETURN 2;
                           ELSE RETURN 1;;
  END IF ATOM;

/* OBJECTS ARE BOTH TUPLES OR BOTH SETS. */
IF TYPE(A) EQ. TUPLE THEN
  (IF NCOMPS(A) NE. NCOMPS(B) THEN RETURN 1;
   ELSE (IF NCOMPS(A) EQ. 0 THEN RETURN 2; ELSE RETURN 3;;)););
/* TYPE MUST BE SETS. */
IF NMEMBS(A) NE. NMEMBS(B) THEN RETURN 1;
ELSE (IF NMEMBS(A) EQ. 0 THEN RETURN 2; ELSE RETURN 4;;);
END EQBASIC;
```

6.8 ELEMENT TEST

```
DEFINEF ELMT(X, S);  /* ENTRY POINT FOR THE SETL X ↠ S. */

/*   THE SETL MEMBERSHIP TEST RESULTS IN AN ERROR EXIT IF
EITHER X IS UNDEFINED, OR IF S IS A NON-STRING ATOM (INCLUDING
OMEGA), OR IF S IS A STRING AND X IS NOT A STRING OF LENGTH
ONE OF THE SAME TYPE AS S.  IN ALL OTHER CASES, EITHER TRUE OR
FALSE IS RETURNED. */

IF X EQ. UNDEFWD THEN ERRTYPE(≠ELMT(X,S), X, UNDEFINED, IS≠,
     X);;
IF TYPE(S) EQ. BOOL  THEN RETURN ELMTBST(X,S);;
IF TYPE(S) EQ. CHAR  THEN RETURN ELMTCST(X,S);;
IF TYPE(S) EQ. TUPLE THEN RETURN ELMTTUP(X,S);;
IF TYPE(S) EQ. SET   THEN RETURN ELMTSET(X,S);;
ERRTYPE(≠ELMT(X,S), S, NOT A SET, TUPLE, NOR STRING, IS≠, S);
END ELMT;



/*                          ELMTBST                          */

DEFINEF ELMTBST(X,S);

/* THIS ROUTINE PERFORMS THE ELEMENT TEST FOR BOOLEAN STRINGS.
S IS KNOWN TO BE A BOOLEAN STRING.  X MUST BE A SINGLE BIT. */

IF TYPE(X) NE. BOOL  OR.  #VALUE(X) NE. 1 THEN
     ERRTYPE(≠ELMT(X,S), S IS A BOOLEAN STRING, BUT X, NOT ≠
          ↠ ≠A BOOLEAN STRING OF LENGTH ONE, IS≠, X); END IF;

(~BIT ↠ VALUE(S)) IF BIT EQ. VALUE(X) THEN RETURN TRUE;;;
RETURN FALSE;
END ELMTBST;
```

```
/*                          ELMTCST                                  */

DEFINEF ELMTCST(X,S);

/* THIS ROUTINE PERFORMS THE ELEMENT TEST FOR CHARACTER
STRINGS.  S IS KNOWN TO BE A CHARACTER STRING.  X MUST BE A
SINGLE CHARACTER. */

IF TYPE(X) NE. CHAR  OR.  #VALUE(X) NE. 1 THEN
     ERRTYPE(#ELMT(X,S).  S IS A CHARACTER STRING, BUT X, #
          * #NOT A CHARACTER STRING OF LENGTH ONE, IS#, X);;

(~C→VALUE(S)) IF C EQ. VALUE(X) THEN RETURN TRUE;;;
RETURN FALSE;
END ELMTCST;



/*                          ELMTTUP                                  */


DEFINEF ELMTTUP(X,T);

/* THIS ROUTINE PERFORMS THE ELEMENT TEST FOR TUPLES,  T IS
KNOWN TO BE A TUPLE.  X MAY BE ANY OBJECT EXCEPT OMEGA (WHICH
WAS CHECKED FOR IN ELMT). */

(~COMP → VALUE(T))
  IF EQUAL(COMP,X) EQ. TRUE THEN RETURN TRUE;; END ~;
RETURN FALSE;
END ELMTTUP;



/*                          ELMTSET                                  */


DEFINEF ELMTSET(X,S);

/* THIS ROUTINE PERFORMS THE MEMBERSHIP TEST FOR SETS.  S IS
KNOWN TO BE A SET,  X MAY BE ANY OBJECT OTHER THAN OMEGA
(WHICH WAS CHECKED FOR BY ELMT). */

IF TYPE(X) EQ. TUPLE THEN RETURN ELSTUP(X,S);
                     ELSE RETURN ELSSMP(X,S);;
END ELMTSET;
```

```
/*                          ELSSMP                                  *

DEFINEF ELSSMP(X,S);

/* THIS ROUTINE IS THE SAME AS ELMTSET, EXCEPT THAT X IS KNOWN
NOT TO BE A TUPLE OF LENGTH ≥ 3. */

IF X EQ. NULLSET THEN RETURN FALSE;;

SEARCH(X, S, M)
  IF EQUAL(X,M) EQ. TRUE THEN RETURN TRUE;;
  ENDSEARCH(M);

/* X IS NOT IN THE SET S. */
RETURN FALSE;
END ELSSMP;



/*                          ELSTUP                                  */

DEFINEF ELSTUP(X,S);

/* THIS ROUTINE IS THE SAME AS ELMTSET, EXCEPT THAT X IS KNOWN
TO BE A TUPLE.
     IF THE TUPLE IS OF LENGTH TWO OR LESS, THEN ELSSMP IS
CALLED.  OTHERWISE, AFTER A CHECK TO SEE IF S IS THE NULL SET,
THE LOGIC IS AS FOLLOWS.  THE GIVEN SET (S) IS SEARCHED FOR A
SPECIAL PAIR THAT BEGINS WITH THE FIRST COMPONENT OF X.  IF
FOUND, THE SECOND COMPONENT OF THE SPECIAL PAIR (WHICH IS A
NON-NULL SET) IS SEARCHED FOR A SPECIAL PAIR THAT BEGINS WITH
THE SECOND COMPONENT OF X.  IF FOUND, THE SEARCH CONTINUES
DEEPER INTO THE STRUCTURE.  IF AT ANY POINT THE SPECIAL PAIR
IS NOT FOUND, THEN EVIDENTLY THE TUPLE IS NOT IN THE SET, AND
A "FALSE" RETURN IS MADE.  IF ALL SPECIAL PAIRS ARE FOUND,
THE SEARCH STOPS AFTER THE (N-2)TH COMPONENT HAS BEEN PRO-
CESSED, WHERE N = ↓VALUE(X).  AT THIS POINT, THE FINAL SET
IN THE CHAIN IS SEARCHED FOR A 2-TUPLE CONTAINING X'S LAST
TWO COMPONENTS, AND A TRUE OR FALSE RETURN IS MADE ACCORDING
TO THE OUTCOME. */

LENGTH = NCOMPS(X);

IF LENGTH LE. 2 THEN RETURN ELSSMP(X,S);;

IF S EQ. NULLSET THEN RETURN FALSE;;
```

```
SET = S;          /* INITIALIZE CURRENT SET. */
I = 1;            /* INITIALIZE COMPONENT NUMBER (OF X). */
ILIM = I + LENGTH - 2;
(WHILE I LT. ILIM DOING I = I + 1;)
  SEARCH((TUP(X))(I), SET, M)
    IF TYPE(M) EQ. SPECPAIR THEN
      IF EQUAL((TUP(X))(I), (VALUE(M))(1)) EQ. TRUE THEN
        /* ADVANCE IN TUPLE. */
        I = I + 1;
        SET = (VALUE(M))(2);
        CONTINUE WHILE I LT. ILIM;
        END IF EQUAL;
      END IF TYPE;
    ENDSEARCH(M);
  /* SEARCH OF SET IS EXHAUSTED. */
  RETURN FALSE;
  END WHILE I;

/* NOW I = ILIM.  CHECK FINAL SET FOR A 2-TUPLE CONTAINING THE
LAST TWO COMPONENTS OF X. */

SEARCH((TUP(X))(I), SET, M)
  IF TYPE(M) EQ. TUPLE  AND.  NCOMPS(M) EQ. 2 THEN
    IF EQUAL((TUP(X))(I), (TUP(M))(1)) THEN
      IF EQUAL((TUP(X))(I+1), (TUP(M))(2)) THEN
        RETURN TRUE;
        END IF EQUAL;
      END IF EQUAL;
    END IF TYPE;
  ENDSEARCH(M);
/* FINAL 2-TUPLE IS NOT IN THE SET. */
RETURN FALSE;
END ELSTUP;
```

6.9 AUGMENT (WITH)

```
DEFINE AUGMENT(X,S);  /* ENTRY POINT FOR S WITH. X. */

/* THIS ROUTINE AUGMENTS S BY ADDING X TO IT.  IT IS EQUIV-
ALENT TO S = S WITH. X,  S IS NOT COPIED UNLESS IT IS THE NULL
SET.  IT IS ASSUMED THAT THE COMPILER WILL CONVERT S WITH. X
TO:

          T = COPY(S);
          CALL AUGMENT(T,X);

IF IT IS NECESSARY TO PRESERVE S.
     NOTE THAT AUGMENT IS A SUBROUTINE (AND NOT A FUNCTION).
S IS BOTH AN INPUT AND AN OUTPUT QUANTITY.  THIS IS TO REMIND
THE USER THAT THE ROUTINE MODIFIES ITS ARGUMENT.
     THE COPY ROUTINE IS NOT INCLUDED IN THIS SPECIFICATION.*/

IF TYPE(S) NE. SET THEN
  ERRTYPE(≠AUGMENT.  S, NOT A SET, IS≠, S);
  END IF;
IF X EQ. UNDEFWD THEN
  ERRTYPE(≠AUGMENT.  X (UNDEFINED) IS≠, X);
  END IF;
AUGAOK(X,S);
RETURN;
END AUGMENT;



/*                          AUGAOK                              */

DEFINE AUGAOK(X,S);

/* THIS ROUTINE IS THE SAME AS AUGMENT, ABOVE, EXCEPT THAT THE
ARGUMENTS ARE NOT VALIDITY-CHECKED. */

IF TYPE(X) EQ. TUPLE THEN AUGTUP(X,S);
ELSE AUGSIMP(X,S);
  END IF;
RETURN;
END AUGAOK;
```

```
/*                          ALGSIMP                              */

DEFINE AUGSIMP(X,S);

/* THIS ROUTINE IS THE SAME AS AUGMENT, ABOVE, EXCEPT THAT THE
FOLLOWING IS KNOWN ABOUT THE ARGUMENTS:
     X IS A VALID SETL OBJECT, AND IS NOT A TUPLE OF LENGTH ≥
       3.  IT MAY BE A SPECIAL PAIR.
     S IS A SET, POSSIBLY NULL.   */

IF S EQ. NULLSET THEN S = SET1SMP(X);  RETURN;  END IF;

HSH = HASH(X);  /* COMPUTE AND SAVE HASH CODE FOR LATER USE.*/
FIRST = (HASHTABLE(S))(HSH//HTSIZE(S) + 1);
/* SEARCH LIST STARTING WITH FIRST. */
M = FIRST;
(WHILE M NE. OM.  DOING M = NEXTM(M);)
  IF EQUAL(M,X) EQ. TRUE THEN
    RETURN;  /* (X IS ALREADY IN THE SET). */
    END IF;
  END WHILE;

/* X IS NOT IN THE SET.  PUT IT IN.  IT IS PLACED AT THE HEAD
OF THE LIST, BECAUSE (1) IT IS SLIGHTLY MORE EFFICIENT TO DO
SO IN THE LITTLE VERSION OF THE RUN TIME LIBRARY, AND (2) A
PROGRAM'S BEHAVIOR IS TYPICALLY LOCAL, SO THIS ITEM MAY BE
REFERENCED BEFORE THE OTHERS. */

NEXTM(X) = FIRST;
(HASHTABLE(S))(HSH//HTSIZE(S) + 1) = X;

/* NOW UPDATE THE HASH CODE OF THE SET, THE NUMBER OF MEMBERS.
AND THE LOAD FACTOR. */

HCODE(S) = ((HCODE(S) AS. BSTRING.) // (HSH AS. BSTRING.))
           AS. INT.;
NMEMBS(S) = NMEMBS(S) + 1;
LOAD(S) = LOAD(S) + 1;

/* EXPAND HASH TABLE IF IT IS TOO DENSELY USED. */

IF TOODENSE(S) THEN EXPAND(S);;
RETURN;
END AUGSIMP;
```

```
DEFINE AUGTUP(X,S);

/* THIS ROUTINE IS THE SAME AS AUGMENT, EXCEPT THAT X IS KNOWN
TO BE A TUPLE (POSSIBLY NULL) AND S IS KNOWN TO BE A SET (ALSO
POSSIBLY NULL).
     THE ROUTINE IS COMPLICATED BECAUSE OF THE WAY TUPLES ARE
BROKEN UP WHEN PUT INTO SETS.  IT WORKS WITHOUT RECURSION, BUT
IT HAS TO USE THE STACK.  OPERATION IS AS FOLLOWS (FOR THE
NON-TRIVIAL CASES).
     FIRST THE GIVEN SET S IS SEARCHED FOR A SPECIAL PAIR THAT
BEGINS WITH THE FIRST COMPONENT OF THE GIVEN TUPLE X.  IF
FOUND, THE SET IS STACKED, AND THE SECOND PART OF THE SPECIAL
PAIR (WHICH IS A SET) IS SEARCHED FOR A SPECIAL PAIR THAT
BEGINS WITH THE SECOND COMPONENT OF THE TUPLE.  IF FOUND, THE
CURRENT SET IS STACKED AND THE PROCESS REPEATS.
     THE SEARCH ENDS WHEN WORKING ON THE INPUT TUPLE COMPONENT
I AND EITHER OF TWO THINGS HAPPENS:
     1.  I = LENGTH OF X - 1, IN WHICH CASE THE CURRENT SET
         IS NOT SEARCHED, OR
     2.  I < LENGTH OF X - 1, BUT NO SPECIAL PAIR THAT BEGINS
         WITH X(I) WAS FOUND.
     IN EITHER CASE, THE CURRENT SET IS AUGMENTED BY THE RE-
MAINING COMPONENTS (FROM I ON).  THIS IS DONE BY CALLING
TUPSPLT, WHICH PROCESSES THE COMPONENTS FROM I ON, AND
RETURNS WITH EITHER A 2-TUPLE (IF I = LENGTH OF X - 1), OR A
SPECIAL PAIR REPRESENTING THE REMAINING COMPONENTS (OF WHICH
THERE ARE THREE OR MORE).  THIS VALUE IS PUT IN THE CURRENT
SET BY AUGSIMPLY.
     NOW A TEST IS MADE TO SEE IF THE NUMBER OF MEMBERS IN THE
CURRENT SET ACTUALLY INCREASED (IT MAY NOT IF I = LENGTH - 1
AND THE 2-TUPLE IS ALREADY IN THE CURRENT SET).  IF THE NUMBER
OF MEMBERS DID NOT INCREASE (WHICH MEANS THAT X WAS ALREADY IN
THE ORIGINAL SET S), THERE IS NOTHING TO DO EXCEPT TO RESTORE
THE STACK AND RETURN.  OTHERWISE ALL THE STACKED SETS HAVE
THEIR MEMBERSHIP COUNT INCREASED BY 1, AND THE STACK IS
RESTORED AND THE ROUTINE RETURNS.
     NOTE THAT THE STACKED SETS DO NOT HAVE THEIR LOADING
FACTOR OR HASH CODE ALTERED.  THE LOADING FACTOR DEFINITELY
SHOULD NOT BE INCREASED.  IT IS OF LITTLE IMPORTANCE WHETHER
OR NOT THE HASH CODE OF THE SET IS ALTERED (PROVIDED OF COURSE
THE RUN TIME LIBRARY IS SELF-CONSISTENT). */

IF NCOMPS(X) LE. 2 THEN AUGSIMP(X,S);  RETURN;  END IF;

IF S EQ. NULLSET THEN S = SETWTH1(X);  RETURN;  END IF;
```

```
/* WE HAVE A NON-TRIVIAL CASE. */

CS = S;    /* INITIALIZE CURRENT SET. */
I = 1;     /* INITIALIZE CURRENT COMPONENT OF X. */
ILIM = NCOMPS(X) - 1;     /* INITIALIZE LAST COMPONENT - 1. */
(WHILE I LT. ILIM)
  COMP = (VALUE(X))(I);  /* GET CURRENT COMPONENT. */

  /* SEARCH CURRENT SET FOR A SPECIAL PAIR THAT BEGINS WITH
  THE CURRENT COMPONENT OF X. */
  SEARCH(COMP, CS, M)
    IF TYPE(M) EQ. SPECPAIR THEN
      IF EQUAL((VALUE(M))(1), COMP) THEN
        /* FOUND THE SPECIAL PAIR.  STACK CS, AND ADVANCE. */
        PUSH(CS);
        I = I + 1;
        CS = (VALUE(M))(2);    /* ADVANCE CURRENT SET. */
        CONTINUE WHILE I LT. ILIM;
        END IF EQUAL;
      END IF TYPE;
    ENDSEARCH(M);    /* ADVANCE IN SET-LIST. */

  /* I < ILIM BUT SEARCH OF CURRENT SET IS EXHAUSTED.  TREAT
  SAME AS I = ILIM CASE (X MUST BE ADDED TO THE SET). */

  QUIT WHILE I LT. ILIM;
  END WHILE I LT. ILIM;

/* EITHER I = ILIM OR I < ILIM BUT THE SPECIAL PAIR THAT
BEGINS WITH COMPONENT I WAS NOT FOUND.  AUGMENT THE CURRENT
SET BY THE REMAINING COMPONENTS, FROM I ON. */

NEWCS = CS;
AUGSIMP(TUPSPLT(X,I,ILIM), NEWCS);
IF NEWCS NE. CS THEN

  /* NUMBER OF MEMBERS HAS INCREASED.  MODIFY THE ORIGINAL SET
  S TO REFLECT NEWCS RATHER THAN CS.  THIS IS DONE BY WORKING
  BACK IN THE STACK, CHANGING EACH STACKED SET.  THE LAST ONE
  BECOMES THE NEW VALUE OF S RETURNED BY THIS ROUTINE. */

  S = NEWCS;
  (WHILE ↓STACK GT. 0)
    POP(NEXTSET);
    I = I - 1;
    S = REPLACE(<SPECPAIR,<(VALUE(X))(I),CS>>, NEXTSET,
                <SPECPAIR,<(VALUE(X))(I),S>>);
    NMEMBS(S) = NMEMBS(S) + 1;
    CS = NEXTSET;
    END WHILE;
```

```
  /* NOW S IS PROPERLY MODIFIED.  NOTE:  THE LITTLE VERSION OF
  SRTL IS BASED ON POINTERS, AND HENCE IT IS ONLY NECESSARY
  UPDATE NMEMBS IN THE STACKED SETS. */
ELSE
  /* NUMBER OF MEMBERS DID NOT INCREASE.  THE TUPLE WAS
  ALREADY IN THE SET.  THIS HAS ALL BEEN A WASTE OF TIME. */

  STACK = NULT.;     /* RESTORE STACK. */
  END IF NEWCS;
RETURN;
END AUGTUP;



/*                          REPLACE                              */


DEFINEF REPLACE(X,S,XNEW);

/* THIS FUNCTION SEARCHES SET S FOR MEMBER X, AND REPLACES IT
WITH XNEW.  X MUST BE PRESENT IN S.  THE ROUTINE HAS NO
COUNTERPART IN THE LITTLE SRTL, BECAUSE LITTLE WORKS WITH
POINTERS AND HENCE CAN MODIFY A MEMBER OF A SET VERY EASILY.
     THE SET RETURNED HAS XNEW IN PLACE OF X AT THE SAME SPOT
IN THE SET-LIST THAT X OCCUPIED.  NO OTHER ALTERATION OF THE
SET IS DONE.  IN PARTICULAR, THE HASH CODE OF THE SET IS NOT
CHANGED, AND THE POSSIBILITY OF A DUPLICATE MEMBER IS NOT
CHECKED FOR.
     THIS ROUTINE IS USED ONLY BY AUGTUP AND DIMTUP TO ALTER
SPECIAL PAIRS IN A SET. */

I = HASH(X)//HTSIZE + 1;
M = (HASHTABLE(S))(I);
PREVMEMB = OM.;      /* INITIALIZE. */
(WHILE M NE. OM. DOING M = NEXTM(M);)
  IF EQUAL(X,M) EQ. TRUE THEN
    /* FOUND X.  REPLACE IT WITH XNEW. */
    /* MAKE NEW MEMBER POINT TO LIST BEYOND X. */
    NEXTM(XNEW) = NEXTM(X);
    /* PUT NEW MEMBER IN LIST IN PLACE OF X. */
    IF PREVMEMB EQ. OM. THEN
      (HASHTABLE(S))(I) = XNEW;
    ELSE
      NEXTM(PREVMEMB) = XNEW;
      END IF;
    /* X AND ITS TAIL WILL BE GARBAGE-COLLECTED. */
    RETURN;
  ELSE
    PREVMEMB = M;
```

```
    END IF EQUAL;
  END WHILE;

ERRIMPL(≠AUGMENT ROUTINE OBTAINED INCONSISTENT RESULTS AS TO ≠
      + ≠WHETHER OR NOT THE OBJECT ≠ + (X AS. CSTRING.)
      + ≠ IS IN THE SET ≠ + (S AS. CSTRING.));

END REPLACE;



/*                          TUPSPLT                                  */


DEFINEF TUPSPLT(T, IMIN, ILIM);

/* THIS FUNCTION HAS AS INPUT A STRING OF N ≥ 2 OBJECTS
STORED IN THE TUPLE T FROM IMIN ≤ I ≤ ILIM + 1.  THE VALUE OF
THE ROUTINE IS EITHER A SPECIAL PAIR (IF N > 2) THAT REPRE-
SENTS THE COMPONENTS, OR IS A 2-TUPLE (IF N = 2) CONTAINING
THE COMPONENTS.
     THE ROUTINE WORKS BY PROCESSING ITS INPUT IN RIGHT-TO-
LEFT ORDER.  FIRST, A 2-TUPLE IS MADE UP CONTAINING THE LAST
TWO COMPONENTS.  THEN A SERIES OF SETS AND SPECIAL PAIRS IS
MADE UP FOR THE OTHER COMPONENTS. */

/* MAKE UP A 2-TUPLE FOR THE LAST TWO COMPONENTS. */
R = <TUPLE, <2, 2, <(VALUE(T))(ILIM), (VALUE(T))(ILIM+1)>>>;

I = ILIM;       /* INITIALIZE FOR LOOP. */
(WHILE I GT. IMIN)
  I = I - 1;    /* POINTS TO NEXT COMPONENT TO PROCESS. */
  /* MAKE UP A SPECIAL PAIR CONTAINING (VALUE(T))(I) AND THE
  SET CONTAINING THE PREVIOUSLY FORMED SPECIAL PAIR (OR 2-
  TUPLE). */
  R = <SPECPAIR, <(VALUE(T))(I), SET1SMP(R)>>;
  END WHILE;
RETURN R;
END TUPSPLT;
```

```
/*                          SETWTH1                                  */

DEFINEF SETWTH1(X);

/* THIS ROUTINE IS GIVEN AN OBJECT S, AND IT RETURNS WITH THE
SET CONTAINING ONLY THAT OBJECT.  NO ARGUMENT CHECKING IS
DONE, AND HENCE IT SHOULD NOT BE CALLED DIRECTLY FROM COMPILED
CODE.
     X MAY BE A LONG TUPLE.
     THIS ROUTINE IS CALLED FROM AUGTUPLE.  */

IF TYPE(X) EQ. TUPLE  AND.  NCOMPS(X) GE. 3 THEN
  RETURN SET1SMP(TUPSPLT(X, 1, NCOMPS(X)-1));
ELSE RETURN SET1SMP(X);;
END SETWTH1;



/*                          SET1SMP                                  */

DEFINEF SET1SMP(X);

/* THIS ROUTINE IS THE SAME AS SETWTH1 EXCEPT THAT X IS
KNOWN NOT TO BE A TUPLE OF LENGTH ≥ 3.
     THIS ROUTINE IS CALLED FROM SETWTH1 AND TUPSPLT. */

HASHORJ = HASH(X);

/* CREATE THE HASH TABLE FOR THE SET.  THE HASH TABLE IS
REPRESENTED BY A SPARSE TUPLE, AND IT MAY HAVE ONLY ONE COM-
PONENT DEFINED.  HENCE #HASHTABLE(S) < HTSIZE(S)  ENERALLY.
THE QUANTITY #HASHTABLE(S) IS NEVER USED IN THIS SPECIF-
ICATION. */

HASHTAB = NULT.;     /* INITIALIZE. */
HASHTAB(HASHORJ//MINHTSIZE + 1) = X;
HASHSET = ((HASHORJ AS. BSTRING.) // (HASHNLS AS. BSTRING.))
          AS. INT.;
RETURN <SET, <1, 1, HASHSET, MINHTSIZE, HASHTAB>>;
END SET1SMP;
```

```
/*                          EXPAND                              */

DEFINE EXPAND(S);

/* THIS SUBRCUTINE DOUBLES THE SIZE OF THE HASH TABLE OF S.
IT IS CALLED BY AUGSIMP WHEN THE HASH TABLE IS TOO DENSELY
POPULATED.
     IF THE NEW SIZE WOULD EXCEED THE MAXIMUM (MAXHTSIZE), NO
ACTION IS TAKEN. */

OLDSIZE = HTSIZE(S);
NEWSIZE = 2*OLDSIZE;
IF NEWSIZE GT. MAXHTSIZE THEN RETURN;;

NEWHASHTAB = NULT.;        /* INITIALIZE. */

/* SCAN MEMBERS OF OLD HASH TABLE, MOVING THEM TO THE NEW HASH
TABLE.  EACH ITEM MUST BE RE-HASHED (TO OBTAIN ONE MORE BIT OF
INFORMATION ON EACH). */

(1 ≤ ~I ≤ OLDSIZE)
  M = (HASHTABLE(S))(I);
  (WHILE M NE. OM.  DOING M = NEXTM(M);)
    NEWI = HASH(M)//NEWSIZE + 1;
    /* PUT M INTO FRONT OF SET-LIST AT NEWI. */
    NEXTM(M) = NEWHASHTAB(NEWI);
    NEWHASHTAB(NEWI) = M;
    END WHILE;
  END ~I;

S = <SET, <NMEMBS(S),LOAD(S),HCODE(S),NEWSIZE,NEWHASHTAB>>;
RETURN;
END EXPAND;
```

6.10 DIMINIS (LESS)

```
DEFINE DIMINIS(X,S);  /* ENTRY POINT FOR S LESS. X. */

/* THIS ROUTINE DIMINISHES S BY REMOVING X FROM IT.  IT IS
EQUIVALENT TO S = S LESS. X.  S IS NOT COPIED; IT IS ASSUMED
THAT THE COMPILER WILL GENERATE A CALL TO THE COPY ROUTINE IF
IT IS NECESSARY TO PRESERVE S. */

IF TYPE(S) NE. SET THEN
  ERRTYPE(≠DIMINIS.  S, NOT A SET, IS≠, S);
  END IF;
IF X EQ. UNDEFWD THEN
  ERRTYPE(≠DIMINIS.  X (UNDEFINED) IS≠, X);
  END IF;

DIMINOK(X,S);
RETURN;
END DIMINIS;



/*                          DIMINOK                              */

DEFINE DIMINCK(X,S);

/* THIS ROUTINE IS THE SAME AS DIMINIS, ABOVE, EXCEPT THAT
THE ARGUMENTS ARE NOT VALIDITY CHECKED. */

IF TYPE(X) EQ. TUPLE THEN DIMTUP(X,S);
ELSE DIMSIMP(X,S);
  END IF;
RETURN;
END DIMINOK;
```

```
/*                          DIMSIMP                                  */

DEFINE DIMSIMP(X,S);

/* THIS ROUTINE IS THE SAME AS DIMINIS, ABOVE, EXCEPT THAT
THE FOLLOWING IS KNOWN ABOUT THE ARGUMENTS:
     X IS A VALID SETL OBJECT, AND IS NOT A TUPLE OF LENGTH ≥
          3.  IT MAY BE A SPECIAL PAIR.
     S IS A SET, POSSIBLY NULL. */

IF S EQ. NULLSET THEN RETURN;;

/* SEARCH FOR X IN SET S, */
HSH = HASH(X);
PREV = OM.;
I = HSH//HTSIZE(S) + 1;
M = (HASHTABLE(S))(I);
(WHILE M NE. OM.  DOING M = NEXTM(M);;)
  IF EQUAL(M,X) EQ. TRUE THEN
    /* FOUND X IN THE SET.   DELETE IT. */
    IF PREV EQ. OM. THEN
      /* FIRST IN LIST,  MOVE NEXT ONE IN. */
      (HASHTABLE(S))(I) = NEXTM(M);
    ELSE  /* NOT FIRST IN LIST.  MAKE PREVIOUS MEMBER POINT
             TO THE FOLLOWING ONE. */
      NEXTM(PREV) = NEXTM(M);
      END IF PREV;
    /* ADJUST NMEMBS, LOAD, AND HASH CODE OF THE SET, */
    NMEMBS(S) = NMEMBS(S) - 1;
    LOAD(S) = LOAD(S) - 1;
    HCODE(S) = ((HCODE(S) AS. BSTRING.) // (HSH AS. BSTRING.))
                AS. INT.;
    /* MAKE THE HASH TABLE SMALLER, IF DESIRABLE, AND RETURN
    WITH THE STANDARD NULL SET IF NMEMBS IS ZERO. */
    IF TOOSPARSE(S) THEN CONTRCT(S);;
    RETURN;
    END IF EQUAL;

  /* RESULT OF EQUAL COMPARISON WAS FALSE.  ADVANCE IN LIST.*/
  END WHILE M;

/* X IS NOT IN THE SET,  NOTHING TO DO. */
RETURN;
END DIMSIMP;
```

```
/*                          DIMTUP                              */

DEFINE DIMTUP(T,S);

/* THIS ROUTINE IS THE SAME AS DIMINIS, EXCEPT THAT T IS
KNOWN TO BE A TUPLE (POSSIBLY NULL), AND S IS KNOWN TO BE A
SET (ALSO POSSIBLY NULL).
     THE ROUTINE IS COMPLICATED BECAUSE OF THE WAY TUPLES ARE
BROKEN UP WHEN PUT INTO SETS.  IT WORKS WITHOUT RECURSION, BUT
IT HAS TO USE THE STACK.  OPERATION IS AS FOLLOWS (FOR THE
NON-TRIVIAL CASES).
     FIRST THE GIVEN SET S IS SEARCHED FOR A SPECIAL PAIR THAT
BEGINS WITH THE FIRST COMPONENT OF THE GIVEN TUPLE T.  IF NOT
FOUND, THEN EVIDENTLY THE TUPLE IS NOT IN THE SET, SO PARA-
METERS ARE RESTORED AND CONTROL RETURNS.  IF THE SPECIAL PAIR
IS FOUND, THEN THE SET AND THE SPECIAL PAIR ARE STACKED, AND
THE SECOND PART OF THE SPECIAL PAIR (WHICH IS A SET) IS
SEARCHED FOR A SPECIAL PAIR THAT BEGINS WITH THE SECOND COM-
PONENT OF T.  IF FOUND, THE CURRENT SET AND SPECIAL PAIRS ARE
STACKED, AND THE PROCESS REPEATS.
     THE SEARCH ENDS AFTER THE (N-2)TH COMPONENT HAS BEEN
SEARCHED AND FOUND TO BE THE FIRST PART OF A SPECIAL PAIR.
THE LAST SPECIAL PAIR AND THE SET CONTAINING IT ARE STACKED,
AND THE CURRENT SET IS MADE TO BE THE SECOND PART OF THE LAST
SPECIAL PAIR.
     NOW THE LAST SET IS SEARCHED FOR A 2-TUPLE CONTAINING THE
INPUT TUPLE'S LAST TWO COMPONENTS.  IF NOT FOUND, PARAMETERS
ARE RESTORED AND CONTROL RETURNS.  IF FOUND, THE 2-TUPLE IS
DELETED FROM THE LAST SET.  IN THE NORMAL CASE, ALL THAT
REMAINS TO BE DONE IS TO DECREMENT THE NUMBER OF MEMBERS AND
THE LOAD FACTOR FROM THE LAST SET, ADJUST ITS HASH CODE, AND
THEN DECREMENT THE NUMBER OF MEMBERS IN ALL STACKED SETS (ALL
THOSE HIGHER UP IN THE CHAIN).
     IN THE EVENT THAT DELETION OF THE 2-TUPLE CAUSES THE LAST
SET TO BECOME NULL, HOWEVER, WE PROCEED AS FOLLOWS (THE POINT
IS THAT WE NEVER LEAVE SPECIAL PAIRS AROUND WHEN THEIR SECOND
COMPONENT IS THE NULL SET).  THE SET AND SPECIAL PAIR AT THE
TOP OF THE STACK ARE OBTAINED, AND THE SPECIAL PAIR IS DELETED
FROM THE SET (USING DIMSIMP).  IF THIS CAUSES THE SET TO
BECOME NULL, THEN THE NEXT SET AND SPECIAL PAIR ARE UNSTACKED,
AND THE PROCESS IS REPEATED.  THIS PROCESS ENDS WHEN EITHER A
DELETION DOES NOT CAUSE THE SET TO BECOME NULL, OR ALL STACKED
SETS HAVE BEEN PROCESSED.  THE ALGORITHM NOW PROCEEDS AS IN
THE NORMAL CASE:  THE REMAINING STACKED SETS (IF ANY) HAVE
THEIR NUMBER OF MEMBERS DECREMENTED (THEY CANNOT BECOME ZERO),
AND CONTROL RETURNS TO THE CALLER.  */
```

```
LENGTH = NCOMPS(T);
IF LENGTH LE. 2 THEN DIMSIMP(T,S);  RETURN;;
IF S EQ. NULLSET THEN RETURN;;

/* WE HAVE A NON-TRIVIAL CASE. */

CS = S;          /* INITIALIZE CURRENT SET. */
ILIM = LENGTH - 1;
(1 ≤ ^I < ILIM)
  /* SEARCH CS FOR A SPECIAL PAIR THAT BEGINS WITH THE CURRENT
  COMPONENT. */
  SEARCH((TUP(T))(I), CS, M)
    IF TYPE(M) EQ. SPECPAIR  AND.
       (EQUAL((TUP(T))(I), (VALUE(M))(1)) EQ. TRUE) THEN
      /* FOUND THE SPECIAL PAIR.  STACK IT AND ITS CONTAINING
      SET, AND ADVANCE. */
      PUSH(<M,CS>);
      I = I + 1;
      CS = (VALUE(M))(2);
      CONTINUE ^I;
      END IF TYPE;
    ENDSEARCH(M);

  /* I < ILIM BUT SEARCH OF CURRENT SET IS EXHAUSTED.  T IS
  NOT IN THE ORIGINAL SET. */
  STACK = NULT.;     /* RESTORE STACK. */
  RETURN;
  END ^I;

/* NOW ALL COMPONENTS < ILIM HAVE BEEN PROCESSED AND CS IS THE
LAST SET IN THE CHAIN.  ALL HIGHER ORDER SETS AND THE SPECIAL
PAIR POINTING TO CS HAVE BEEN STACKED.  NOW DELETE THE 2-TUPLE
CONTAINING THE LAST TWO COMPONENTS FROM THIS SET.  THIS IS
DONE BY FORMING AN APPROPRIATE 2-TUPLE, AND USING DIMSIMP.
THIS IS RATHER INEFFICIENT, AND ANYONE WHO IS CONCERNED ABOUT
IT IS INVITED TO COPY THE BULK OF DIMSIMP AT THIS POINT. */

NEWCS = CS;
DIMSIMP(<TUPLE, <2,2,<(TUP(T))(ILIM),(TUP(T))(ILIM+1)>>>,
                                                    NEWCS);
IF NEWCS EQ. CS THEN /* NOTHING WAS DELETED. */
  STACK = NULT;       /* RESTORE THE STACK. */
  RETURN;
  END IF;
```

```
/* SOMETHING WAS DELETED.  IF THE SET BECAME NULL, DELETE THE
SPECIAL PAIR AT THE TOP OF THE STACK FROM THE SET AT THE TOP
OF THE STACK, POP THE STACK, AND CONTINUE UNTIL A NON-NULL SET
REMAINS. */

(WHILE NEWCS EQ. NULLSET)
  /* IF STACK IS EMPTY, RETURN WITH S = NULLSET. */
  IF STACK EQ. NULT. THEN S = NULLSET;  RETURN; END IF;
  POP(<SP, NEWCS>);
  DIMSIMP(SP, NEWCS);
  END WHILE;

/* NOW NEWCS IS NON-NULL,  CONTINUE WORKING BACK IN THE STACK,
DECREMENTING NMEMBS IN ALL STACKED SETS. */

(WHILE STACK NE. NULT.)
  POP(<SP, S>);
  S = REPLACE(SP, S, <SPECPAIR,<(VALUE(SP))(1),NEWCS>>);
  NMEMBS(S) = NMEMBS(S) - 1;
  NEWCS = S;
  END WHILE;

/* DONE. */

RETURN;
END DIMTUP;
```

```
DEFINE CONTRCT(S);

/* THIS ROUTINE REDUCES THE SIZE OF THE GIVEN SET'S HASH
TABLE, IF APPROPRIATE, BY A FACTOR OF ONE HALF.  IT IS CALLED
BY DIMSIMP AND DIMFAOK WHEN IT IS FOUND THAT THE SET'S
HASH TABLE IS TOO SPARSELY USED.
     THE LOGIC IS AS FOLLOWS:  IF THE SET HAS NO MEMBERS, THE
STANDARD NULL SET IS RETURNED.  IF THE SET HAS MEMBERS BUT
HALF ITS HASH TABLE SIZE IS LESS THAN THE MINUMUM (MINHTSIZE),
THEN NO ACTION IS TAKEN.  OTHERWISE THE REDUCTION IS DONE, IN
PLACE.  LISTS IN THE LOWER HALF OF THE HASH TABLE ARE COMBINED
WITH THOSE IN THE UPPER HALF. */

IF NMEMBS(S) EQ. 0 THEN S = NULLSET;  RETURN;;

NEWSIZE = HTSIZE(S)/2;
IF NEWSIZE LT. MINHTSIZE THEN RETURN;;

(1 ≤ 'I ≤ NEWSIZE)
  /* GET OBJECT IN LOWER HALF OF HASH TABLE (PLUS ITS LIST).*/
  OBJECT = (HASHTABLE(S))(I+NEWSIZE);
  IF OBJECT EQ. OM. THEN CONTINUE 'I;
  /* INSERT OBJECT (INCLUDING THE LIST IT POINTS TO) AT THE
  TAIL END OF THE CORRESPONDING LIST AT POSITION I. */
  IF (HASHTABLE(S))(I) EQ. OM. THEN /* VACUOUS. */
    (HASHTABLE(S))(I) = OBJECT;     /* PUT WHOLE LIST IN. */
  ELSE  /* SEARCH LIST AT I FOR ITS TAIL END. */
    L = (HASHTABLE(S))(I);
    M = NEXTM(L);
    (WHILE M NE. OM.)
      L = M;
      M = NEXTM(M);
      END WHILE;
    /* NOW L IS THE LAST MEMBER IN THE LIST. */
    NEXTM(L) = OBJECT;   /* ATTACH LIST FROM BOTTOM HALF OF
                            HASH TABLE. */
    END IF;
  END 'I;

/* THE FOLLOWING IS DONE IN-PLACE IN THE LITTLE CODE. */
S = <SET, <NMEMBS(S), LOAD(S), HCODE(S), NEWSIZE,
          (HASHTABLE(S))(1:NEWSIZE)>>;
RETURN;
END CONTRCT;
```

6.11 DIMF (LESF)

```
DEFINE DIMF(X,S);  /* ENTRY POINT FOR S LESF. X. */

/* THIS SUBRCUTINE DIMINISHES S BY DELETING FROM IT ALL TUPLES
OF LENGTH ≥ 2 THAT BEGIN WITH X.  IT IS EQUIVALENT TO S =
S LESF. X. */

IF TYPE(S) NE. SET THEN
  ERRTYPE(≠DIMF.  S. NOT A SET, IS≠, S);
  END IF;

DIMFAOK(X,S);  /* NOTE:  X = UNDEFWD IS OK. */
RETURN;
END DIMF;
```

```
/*                          DIMFAOK                              */

DEFINE DIMFAOK(X,S);

/* THIS SUBROUTINE IS THE SAME AS DIMF, ABOVE, EXCEPT
THAT S IS KNOWN TO BE A SET.  X MAY BE UNDEFINED. */

IF S EQ. NULLSET THEN RETURN;;

HSH = HASH(X);
PREV = OM.;     /* INITIALIZE. */
I = HSH//HTSIZE(S) + 1;
M = (HASHTABLE(S))(I);
(WHILE M NE. OM.  DOING M = NEXTM(M);)
  FIRSTCOMP = 0;     /* INITIALIZE TO AN IMPOSSIBLE OBJECT
                        (NOT A PAIR). */
  IF TYPE(M) EQ. TUPLE  AND.  NCOMPS(M) EQ. 2 THEN
    FIRSTCOMP = (TUP(M))(1);
  ELSE IF TYPE(M) EQ. SPECPAIR THEN
    FIRSTCOMP = (VALUE(M))(1);;

  IF EQUAL(FIRSTCOMP,X) EQ. TRUE THEN
    /* FOUND AN OBJECT TO BE DELETED. */
    IF PREV EQ. OM. THEN (HASHTABLE(S))(I) = NEXTM(M);
                    ELSE NEXTM(PREV) = NEXTM(M);;
    IF TYPE(M) EQ. TUPLE THEN NBRDELETED = 1;
                         ELSE NBRDELETED = NMEMBS((VALUE(M))(2));;
    NMEMBS(S) = NMEMBS(S) - NBRDELETED;
    LOAD(S) = LOAD(S) - 1;
    HCODE(S) = ((HCODE(S) AS. BSTRING.)//(HSH AS. BSTRING.))
               AS. INT.;
  ELSE /* DO NOT DELETE IT. */
    PREV = M;
    END IF EQUAL;
  END WHILE;

/* COMPRESS HASH TABLE IF NECESSARY. */
IF NMEMBS(S) EQ. 0 THEN S = NULLSET;
ELSE
  (WHILE TOOSPARSE(S) AND. HTSIZE(S) GT. MINHTSIZE)
    CONTRCT(S);
    END WHILE;
  END IF;
RETURN;
END DIMFAOK;
```

6.12 ARBITRARY ELEMENT

```
DEFINEF ARB(S);  /* ENTRY POINT FOR THE SETL AN.S (ANY S). */

IF TYPE(S) EQ. SET   THEN RETURN ARBSET(S);;
IF TYPE(S) EQ. TUPLE THEN RETURN ARBTUP(S);;
IF TYPE(S) EQ. CHAR  THEN RETURN ARBCSTR(S);;
IF TYPE(S) EQ. BOOL  THEN RETURN ARBBSTR(S);;

ERRTYPE(#ARB(S).  S. NOT A SET, TUPLE. NOR STRING. IS#, S);
END ARB;



/*                              ARBBSTR                                 */


DEFINEF ARBBSTR(S);

/* ARB ROUTINE FOR BIT STRINGS. */

IF S EQ. NULLBSTR THEN RETURN UNDEFWD;
ELSE RETURN (VALUE(S))(1);;  /* (ANY BIT COULD BE USED.) */
END ARBBSTR;



/*                              ARBCSTR                                 */


DEFINEF ARBCSTR(S);

/* ARB ROUTINE FOR CHARACTER STRINGS. */

IF S EQ. NULLCSTR THEN RETURN UNDEFWD;
ELSE RETURN (VALUE(S))(1);; /*(ANY CHARACTER COULD BE USED,)*/
END ARBCSTR;
```

```
/*                          ARBTUP                              */

DEFINEF ARBTUP(T);

/* ARB ROUTINE FOR TUPLES. */

/* THE SETL DEFINITION IS ANY DEFINED COMPONENT (IF THERE IS
ONE). */

LENGTH = NCOMPS(T);
IF LENGTH EQ. 0 THEN RETURN UNDEFWD;
                ELSE RETURN (VALUE(T))(LENGTH);
END ARBTUP;



/*                          ARBSET                              */

DEFINEF ARBSET(S);

/* THE RESULT IS THE FIRST MEMBER ENCOUNTERED IN A LINEAR SCAN
OF THE HASH TABLE, AND UNDEFINED IF S IS NULL. */

IF S EQ. NULLSET THEN RETURN UNDEFWD;;

MEMB = ARBSIMP(S);
IF TYPE(MEMB) NE. SPECPAIR THEN RETURN MEMB;;
/* OBTAINED A SPECIAL PAIR.  MUST BUILD A TUPLE (OF LENGTH ≥
3) FROM IT. */

NEWTUPLE = NULLTUPLE;
(WHILE TYPE(MEMB) EQ. SPECPAIR)
  /* ADD FIRST COMPONENT OF SPECIAL PAIR ON AS LAST COMP. */
  TUPADD1(NEWTUPLE, (VALUE(MEMB))(1));
  MEMB = ARBSIMP((VALUE(MEMB))(2));  /* GET NEXT SPECIAL
                                        PAIR OR 2-TUPLE. */
  END WHILE;

/* NOW MEMB MUST BE A 2-TUPLE.  ATTACH ITS TWO COMPONENTS TO
THE TUPLE BEING BUILT. */
CONCATT(NEWTUPLE, MEMB);

RETURN NEWTUPLE;
END ARBSET;
```

```
/*                          ARBSIMP                                  */

DEFINEF ARBSIMP(S);

/* THIS FUNCTION RETURNS WITH AN ARBITRARY MEMBER OF S.  S IS
KNOWN TO BE NON-NULL.  THE RESULT MAY BE A SPECIAL PAIR, IN
WHICH CASE A CORRESPONDING TUPLE IS NOT BUILT UP. */

(1 ≤ ~I ≤ HTSIZE(S))
  MEMB = (HASHTABLE(S))(I);
  IF MEMB NE. OM. THEN RETURN MEMB(1:2);
  /* (THE DELETION OF MEMB(3) IS NECESSARY). */
  END ~I;

ERRIMPL(≠ARBSET ROUTINE HAS ENCOUNTERED INCONSISTENT RESULTS ≠
      + ≠AS TO WHETHER OR NOT THE SET ≠ + (S AS. CSTRING.)
      + ≠ IS NULL.≠);
END ARBSIMP;
```

6.13 ITERATION

THE ROUTINES IN THIS SECTION IMPLEMENT THE SETL ITERATION ∪X → S. THE NUMERICAL ITERATION, M ≤ ∪I ≤ N, IS NOT INCLUDED IN THIS SPECIFICATION, AS IT IS ASSUMED THAT NUMERICAL ITERATION WILL BE IMPLEMENTED BY COMPILER-GENERATED CALLS TO ARITHMETIC ROUTINES.

S IN THE ITERATION ∪X → S MAY BE A STRING, TUPLE, OR SET (SEE SETL NEWSLETTER 42 PAGE 4). THE PRIMARY ITERATION ROUTINE IS ≠NEXT≠ BELOW. OTHER ROUTINES ARE NEXTBIT, NEXTCHR, NEXTCMP, AND NEXTMEM, CORRESPONDING TO THE TYPE OF S.

ALL THE ITERATION ROUTINES TAKE TWO ARGUMENTS: S AND A ≠CONTROL ITEM≠ C. ON FIRST ENTRY, THE CONTROL ITEM (WHICH WOULD GENERALLY BE HELD IN A COMPILER-GENERATED TEMPORARY) MUST BE THE INTEGER ZERO. SUBSEQUENTLY, THE CONTROL ITEM SHOULD NOT BE ALTERED BY ANY ROUTINE EXCEPT ≠NEXT≠ (≠NEXT≠ ALTERS ITS FIRST PARAMETER). THE COMPLETION OF ITERATION IS SIGNALED BY RETURNING AN UNDEFINED VALUE BY ≠NEXT≠.

THE CONTROL ITEM IS A COMPLEX DATA AGGREGATE IN THE CASE OF ITERATING OVER A SET THAT CONTAINS LONG TUPLES (SEE NEXTMEM BELOW).

THE COMPILER MAY TRANSLATE THE ITERATION

(∪X → S | COND) BLOCK

AS ILLUSTRATED BELOW.

```
        T = 0;
/BACK/  X = NEXT(T,S);
        IF (X .EQ. UNDEFWD) GO TO OUT;
        IF (COND) THEN BLOCK ENDIF;
        GO TO BACK;
/OUT/   ;
```

```
/*                          NEXT                                    */

DEFINEF NEXT(C,S);  /* ENTRY POINT FOR ITERATION, "X ← S, */

/* GIVEN A CONTROL ITEM C AND S, THIS ROUTINE FINDS THE ≠NEXT≠
OBJECT IN S.  IF C IS ZERO, IT RETURNS WITH THE ≠FIRST≠
OBJECT.  IF C INDICATES THE ≠LAST≠ OBJECT, IT RETURNS WITH
THE RESULT UNDEFINED. */

IF TYPE(S) EQ. SET   THEN RETURN NEXTMEM(C,S);;
IF TYPE(S) EQ. TUPLE THEN RETURN NEXTCMP(C,S);;
IF TYPE(S) EQ. CHAR  THEN RETURN NEXTCHR(C,S);;
IF TYPE(S) EQ. BOOL  THEN RETURN NEXTBIT(C,S);;

ERRTYPE(≠NEXT(C,S), S, NOT A SET, TUPLE, NOR STRING, IS≠, S);
END NEXT;



/*                          NEXTBIT                                 */

DEFINEF NEXTBIT(C,S);

/* THIS ROUTINE EXTRACTS THE NEXT BIT FROM BOOLEAN STRING S
AFTER THE ONE INDICATED BY C.  C IS THE BIT INDEX IN S OF THE
BIT RETURNED. */

IF ↓VALUE(S) EQ. C THEN RETURN UNDEFWD;;
C = C + 1;      /* INCREMENT BIT INDEX. */
RETURN <BOOL, (VALUE(S))(C)>;
END NEXTBIT;
```

```
/*                          NEXTCHR                                  */

DEFINEF NEXTCHR(C,S);

/* THIS ROUTINE EXTRACTS THE NEXT CHARACTER FROM CHARACTER
STRING S AFTER THE ONE INDICATED BY C.  C IS THE CHARACTER
INDEX OF THE CHARACTER RETURNED. */

IF ↓VALUE(S) EQ. C THEN RETURN UNDEFWD;;
C = C + 1;      /* INCREMENT CHARACTER INDEX. */
RETURN <CHAR, (VALUE(S))(C)>;
END NEXTCHR;



/*                          NEXTCMP                                  */

DEFINEF NEXTCMP(C,T);

/* THIS ROUTINE EXTRACTS THE NEXT DEFINED COMPONENT FROM TUPLE
T AFTER THE ONE INDICATED BY C.  C IS AN INDEX INTO T. */

(WHILE C LT. NCOMPS(T))  /* LOOP FOR SKIPPING UNDEF. COMPS. */
  C = C + 1;
  COMP = (TUP(T))(C);
  IF COMP NE. UNDEFWD THEN RETURN COMP;;
  END WHILE;

/* C = NCOMPS(T).  LAST COMPONENT WAS ALREADY RETURNED. */
RETURN UNDEFWD;
END NEXTCMP;
```

```
DEFINEF NEXTMEM(C,S);

/* THIS ROUTINE GETS THE NEXT MEMBER FROM SET S AFTER THE ONE
INDICATED BY C.  THE ROUTINE IS COMPLICATED BECAUSE OF THE WAY
LONG TUPLES ARE STORED IN SETS.
     THE MAIN FEATURES OF THIS ALGORITHM HAVE TO DO WITH
EFFICIENTLY ITERATING OVER SETS CONTAINING LONG TUPLES.  TWO
SIGNIFICANT POINTS ARE:

1.  THE TUPLE RETURNED IS NOT REBUILT FROM SCRATCH EACH TIME.
    INSTEAD, ONLY THE LAST FEW COMPONENTS ARE CHANGED, AS
    REQUIRED.

2.  THE ROUTINE DOES NOT BEGIN ITS PROCESSING WITH THE INPUT
    SET S (EXCEPT ON FIRST ENTRY).  INSTEAD, IT QUICKLY
    LOCATES, FROM C, THE CURRENT SET BEING ITERATED OVER.

     THE CONTROL ITEM C IS BOTH AN INPUT AND AN OUTPUT PARA-
METER.  ITS FORMAT IS:

                C = <OLDM, A>,

WHERE OLDM IS THE LAST MEMBER RETURNED (SUCH AS AN N-TUPLE),
AND A IS THE *ADDRESS* OF OLDM.  A IS IN GENERAL SOMEWHAT
INVOLVED STRUCTURE OF THE FOLLOWING FORM:

            A = <INDEX, CURSET, MEMB, LINK>.

THE ADDRESS IS A 4-TUPLE WHOSE LAST COMPONENT (LINK) IS AN
ADDRESS (ANOTHER 4-TUPLE).  THE INNERMOST ADDRESS IS A 4-TUPLE
WHOSE LAST COMPONENT IS THE INTEGER ZERO.  INDEX IS THE
CURRENT HASH TABLE INDEX.  CURSET IS THE CURRENT SET BEING
ITERATED OVER.  MEMB IS THE CURRENT MEMBER.  IT IS USED TO
DEFINE THE NEXT MEMBER VIA NEXTM, AND TO PROVIDE INFORMATION
AS TO JUST HOW TO MODIFY OLDM.  THE OUTERMOST A-STRUCTURE
REFERS TO THE INNERMOST SET IN THE CHAIN OF SETS USED TO
REPRESENT TUPLES IN SETS.
     THE OVERALL STRATEGY OF THE ROUTINE, ONCE IT GETS GOING,
IS:

1.  IF POSSIBLE, GET THE NEXT MEMBER AFTER MEMB, USE IT TO
    SUPPLY THE NEW LAST TWO COMPONENTS OF OLDM, AND RETURN
    OLDM.

2.  OTHERWISE, INCREMENT INDEX TO SEARCH FOR A NEW MEMBER.

3.  IF INDEX > HTSIZE(CURSET), *BACK UP* BY SETTING A = LINK,
```

```
      AND CONTINUE ITERATING OVER THE PREVIOUS SET.

   4. IF AT ANY TIME A SPECIAL PAIR IS OBTAINED FOR MEMB,
      ≠ADVANCE≠ BY SETTING A = <0, MEMB(2), NULL., A>, ADDING
      MEMB(1) ON TO OLDM, AND ITERATING OVER SET MEMB(2).

   THE END OF THE WHOLE ITERATION IS SIGNIFIED BY ATTEMPTING TO
   BACK UP BUT FINDING THAT LINK IS ZERO.
        HOUSEKEEPING DETAILS, SUCH AS PROPERLY MODIFYING THE
   RIGHT-HAND END OF OLDM, ARE MESSY.
        THE LITTLE VERSION WORKS WITH POINTERS, AND A IS A CHAIN
   OF SMALL BLOCKS, EACH CONTAINING AN INDEX, A POINTER TO A SET,
   A MEMBER ROOT WORD, AND A LINK POINTER. */


   /* INITIALIZE A AND, IF NOT FIRST ENTRY, OLDM. */
   IF C .NE. 0 THEN OLDM = C(1);
                    A = C(2);
               ELSE /* FIRST ENTRY. */
                    A = <0, S, NULL., 0>;
                 END IF C;

   /*GET THE NEXT MEMBER AFTER THE ONE INDICATED BY A. */

GETM:  M = NEXTM(MEMB(A));

TESTM: IF M EQ. OM. THEN
         /*ADVANCE TO NEXT SLOT IN HASH TABLE. */
         INDEX(A) = INDEX(A) + 1;
         IF INDEX(A) GT. HTSIZE(CURSET(A)) THEN
           /* THROUGH WITH CURRENT SET.  BACK UP TO PREVIOUS ONE,
           IF IT EXISTS. */
           IF LINK(A) EQ. 0 THEN  /* DONE. */ RETURN UNDEFWD;;
           /* DELETE LAST COMPONENT(S) FROM OLDM. */
           IF TYPE(MEMB(A)) EQ. TUPLE THEN L = 2; ELSE L = 1;;
           NCOMPS(OLDM) = NCOMPS(OLDM) - L;
           A = LINK(A);  /* BACK UP A. */
           GO TO GETM;
         ELSE /* INDEX(A) ≤ HTSIZE. */
           M = (HASHTABLE(CURSET(A)))(INDEX(A));
           GO TO TESTM;
           END IF INDEX(A);
         END IF M;
```

```
/* GOT A MEMBER M TO PROCESS. */

IF LINK(A) EQ. 0 THEN
  /* WE ARE AT FIRST LEVEL SET. */
  MEMB(A) = M; /* SAVE M FOR NEXT ENTRY. */
  IF TYPE(M) EQ. SPECPAIR THEN

    /* FIRST LEVEL SET AND M IS A SPECIAL PAIR.  SET OLDM
    AND ADVANCE A. (NOTE: THE CURRENT OLDM MAY NOT EVEN
    BE A TUPLE, HENCE ALL OF OLDM MUST BE SET). */

    OLDM = <TUPLE, <1, 3, <(VALUE(M))(1)>>>;
    A = <0, (VALUE(M))(2), NULT., A>;
    GO TO GETM;

  ELSE
    /* FIRST LEVEL SET AND M IS NOT A SPECIAL PAIR (SIMPLE
    CASE). */

    C = <M, A>;  /* SET C TO REFLECT NEW M AND A. */
    RETURN M;
    END IF TYPE;
  END IF LINK(A);


/* LINK(A) ≠ 0; WE ARE NOT AT FIRST LEVEL SET. */

IF TYPE(M) EQ. SPECPAIR THEN

  /* NOT FIRST LEVEL SET AND M IS A SPECIAL PAIR.  PUT M(1)
  INTO OLDM (WHICH MUST BE A TUPLE) AND ADVANCE A.  IF THIS IS
  THE FIRST ENCOUNTER OF A SPECIAL PAIR AT THIS LEVEL, CONCAT-
  ENATE M(1) TO OLDM,  OTHERWISE REPLACE LAST ONE OR TWO COM-
  PONENTS OF OLDM WITH M(1). */

  IF MEMB(A) EQ. NULT. THEN TUPADD1(OLDM, (VALUE(M))(1));
  ELSE
    L = NCOMPS(OLDM);
    IF TYPE(MEMB(A)) EQ. SPECPAIR THEN
      (TUP(OLDM))(L) = (VALUE(M))(1);
    ELSE /* MEMB(A) MUST BE A 2-TUPLE. */
      (TUP(OLDM))(L-1) = (VALUE(M))(1);
      NCOMPS(OLDM) = NCOMPS(OLDM) - 1;
      END IF TYPE;
    END IF MEMB(A);
  MEMB(A) = M;  /* UPDATE ADDRESS AT CURRENT LEVEL. */
  A = <0, (VALUE(M))(2), NULT., A>;  /* ADVANCE. */
  GO TO GETM;
```

```
ELSE /* M MUST BE A 2-TUPLE.  ADD ITS TWO COMPS. TO OLDM. */

  IF MEMB(A) EQ. NULT. THEN CONCATT(OLDM, M);
  ELSE
    L = NCOMPS(OLDM);
    IF TYPE(MEMB(A)) EQ. SPECPAIR THEN
      NCOMPS(OLDM) = NCOMPS(OLDM) - 1;
      CONCATT(OLDM, M);
    ELSE /* MEMB(A) MUST BE A 2-TUPLE. */
      (TUP(OLDM))(L-1) = (TUP(M))(1);
      (TUP(OLDM))(L) = (TUP(M))(2);
      END IF TYPE;
    END IF MEMB(A);
  MEMB(A) = M; /* UPDATE ADDRESS AT CURRENT LEVEL. */
  C = <OLDM, A>;
  RETURN OLDM;
  END IF TYPE(M);


/* AUXILIARY ROUTINES FOR ACCESSING ADDRESS TUPLE (THESE
MAY BE USED IN SINISTER MODE). */

DEFINEF INDEX(A);   RETURN A(1); END;
DEFINEF CURSET(A);  RETURN A(2); END;
DEFINEF MEMB(A);    RETURN A(3); END;
DEFINEF LINK(A);    RETURN A(4); END;


END NEXTMEM;
```

6.14 FUNCTIONAL APPLICATION, RETRIEVAL

THE ROUTINES IN THIS SECTION IMPLEMENT SETL FUNCTIONAL APPLICATION FOR RIGHT HAND SIDE EXPRESSIONS. ROUTINE NAMES ARE:

OF	F(X), MAIN ROUTINE
OFBSTR	F(X), F A BOOLEAN STRING
OFCSTR	F(X), F A CHARACTER STRING
OFTUPLE	F(X), F A TUPLE
OFSET	F(X), F A SET
OFN	F(X1, ..., XN)
OFA	F⌊X⌋
OFAN	F⌊X1, ..., XN⌋
OFB	F{S}, MAIN ROUTINE
OFBBOOL	F{S}, S A BOOLEAN STRING
OFBCHAR	F{S}, S A CHARACTER STRING
OFBTUP	F{T}, T A TUPLE
OFBSET	F{S}, S A SET
OFBN	F{S1, ..., SN}

```
DEFINEF OF(F, X);

/* THIS IS THE MAIN ROUTINE FOR F(X), FOR X A SINGLE ITEM,
FOR RETRIEVAL. */

IF TYPE(F) EQ. SET   THEN RETURN OFSET(F, X);;
IF TYPE(F) EQ. TUPLE THEN RETURN OFTUPLE(F, X);;
IF TYPE(F) EQ. CHAR  THEN RETURN OFCSTR(F, X);;
IF TYPE(F) EQ. BOOL  THEN RETURN OFBSTR(F, X);;
IF TYPE(F) EQ. FUN   THEN
  ERRMSG(≠COMPILER ERROR OR ATTEMPT TO USE A FEATURE NOT YET ≠
     + ≠IMPLEMENTED.  THE ROUTINE FOR F(X) (OF) WAS CALLED ≠
     + ≠WITH F A PROCEDURE.≠);
  RETURN;;
ERRTYPE(≠F(X) (OF), F IS NEITHER A SET, TUPLE, NOR STRING.  ≠
     + ≠F IS≠, F);
END OF;
```

```
/*                          OFBSTR                                  */

DEFINEF OFBSTR(F, X);

/* THIS ROUTINE EVALUATES F(X) FOR F A BOOLEAN STRING. */

IF TYPE(X) EQ. INT THEN
  IF VALUE(X) GT. 0 THEN
    IF VALUE(X) LE. ↓VALUE(F) THEN
      RETURN <BOOL, (VALUE(F))(VALUE(X))>;;
    END IF VALUE;
  END IF TYPE;

/* EITHER X IS NOT AN INTEGER OR IT IS OUT OF RANGE. */

IF TYPE(X) EQ. INT THEN /* OUT OF RANGE. */
  RETURN UNDEFWD;
ELSE  /* NOT AN INTEGER. */
  ERRTYPE(≠F(X) (OF), F IS A BOOLEAN STRING, X IS≠, X);
  END IF;
END OFBSTR;



/*                          OFCSTR                                  */

DEFINEF OFCSTR(F, X);

/* THIS ROUTINE EVALUATES F(X) FOR F A CHARACTER STRING. */

IF TYPE(X) EQ. INT THEN
  IF VALUE(X) GT. 0 THEN
    IF VALUE(X) LE. ↓VALUE(F) THEN
      RETURN <CHAR, (VALUE(F))(VALUE(X))>;;
    END IF VALUE;
  END IF TYPE;

/* EITHER X IS NOT AN INTEGER OR IT IS OUT OF RANGE. */

IF TYPE(X) EQ. INT THEN /* OUT OF RANGE. */
  RETURN UNDEFWD;
ELSE /* NOT AN INTEGER. */
  ERRTYPE(≠F(X) (OF), F IS A CHARACTER STRING, X IS≠, X);
  END IF;
END OFCSTR;
```

```
DEFINEF OFTUPLE(F, X);

/* THIS ROUTINE EVALUATES F(X) FOR F A TUPLE. */

IF TYPE(X) EQ. INT THEN
  IF VALUE(X) GT. 0 THEN
    IF VALUE(X) LE. NCOMPS(F) THEN
      RETURN (TUP(F))(VALUE(X));
    END IF VALUE;
  END IF TYPE;

/* X IS NOT AN INTEGER OR IS OUT OF RANGE. */

IF TYPE(X) EQ. INT THEN
  /* X IS OUT OF RANGE.  F MAY BE NULL. */
  RETURN UNDEFWD;
ELSE  /* X IS NOT AN INTEGER. */
  ERRTYPE(#F(X) (OF), F IS A TUPLE, X IS#. X);
  END IF;
END OFTUPLE;
```

```
DEFINEF OFSET(F, X);

/* F IS KNOWN TO BE A SET.  X MAY BE ANYTHING EXCEPT
UNDEFINED.
     THE ROUTINE SEARCHES THE SET F FOR A TUPLE OF LENGTH
≥ 2 BEGINNING WITH X.  NULL TUPLES AND TUPLES OF LENGTH ONE
ARE SKIPPED OVER.  IF A TUPLE OF LENGTH TWO IS FOUND, THE
RESULT IS ITS SECOND COMPONENT.  A TUPLE OF LENGTH ≥ 3 IS
STORED AS A SPECIAL PAIR.  IF A SPECIAL PAIR IS FOUND
BEGINNING WITH X, THEN ITS SET (THE SET OF TAILS OF ALL
TUPLES IN F THAT BEGIN WITH X AND ARE OF LENGTH ≥ 3) IS
EXAMINED.  IF THIS SET CONTAINS EXACTLY ONE MEMBER, THEN IT
IS EXTRACTED (USING THE ARBSET ROUTINE) AS THE RESULT.  IF
IT CONTAINS MORE THAN ONE MEMBER (IT CANNOT BE NULL) THEN
EVIDENTLY THE ORIGINAL SET HAD MORE THAN ONE TUPLE BEGINNING
WITH X, SO THE RESULT OF F(X) IS UNDEFINED.
     AFTER FINDING THE RESULT, THE SEARCH OF F MUST CONTINUE
TO SEE IF THE IMAGE OF X IS MULTIPLY DEFINED, IN WHICH CASE
THE RESULT IS UNDEFINED.  A SWITCH CALLED #DEFINED# IS USED
IN CONNECTION WITH THIS. */

IF X EQ. UNDEFWD THEN
  ERRTYPE(#F(X) (OF), F IS A SET, X IS UNDEFINED:#, X);;

IF F EQ. NULLSET THEN RETURN UNDEFWD;;

DEFINED = FALSE.;          /* INITIALIZE MULTIPLY DEFINED SW, */
SEARCH(X, F, M);
  IF TYPE(M) EQ. TUPLE THEN
    IF NCOMPS(M) EQ. 2 THEN
      IF EQUAL((TUP(M))(1), X) EQ. TRUE THEN
        /* FOUND A 2-TUPLE BEGINNING WITH X. */
        IF DEFINED THEN
          /* MULTIPLY DEFINED. */
          RETURN UNDEFWD;  END IF DEFINED;
        /* GOT A FIRST RESULT. */
        DEFINED = TRUE.;
        ANSWER = (TUP(M))(2);
        END IF EQUAL;
      END IF NCOMPS;
  ELSE  /* NOT A TUPLE.  CHECK FOR SPECIAL PAIR. */
    IF TYPE(M) EQ. SPECPAIR THEN
      IF EQUAL((VALUE(M))(1), X) EQ. TRUE THEN
        /* FOUND AN N-TUPLE BEGINNING WITH X. */
        IF DEFINED OR. NMEMBS((VALUE(M))(2)) GT. 1 THEN
          /* MULTIPLY DEFINED. */
          RETURN UNDEFWD;  END IF DEFINED;
```

```
        /* GOT A FIRST RESULT. */
        DEFINED = TRUE.;

        /* EXTRACT THE UNIQUE MEMBER FROM M(2).  NOTE THAT
        THIS IS A COMPLICATED OPERATION IF THE MEMBER IS A
        TUPLE OF LENGTH ≥ 3. */

        ANSWER = ARBSET((VALUE(M))(2));
        END IF EQUAL;
      END IF TYPE(M);
    END IF TYPE(M);
  ENDSEARCH(M);

/* SEARCH IS COMPLETE.  AT THIS POINT, IF DEFINED = FALSE.,
NO VALUE OF F(X) WAS FOUND.  IF DEFINED = TRUE., EXACTLY ONE
VALUE WAS FOUND. */

IF DEFINED THEN RETURN ANSWER;
           ELSE RETURN UNDEFWD;;
END OFSET;



/*                          OFN                              */


DEFINEF OFN(F, X);

/* THIS ROUTINE EVALUATES F(X1, X2, ..., XN).  F MUST BE A SET
AND X MUST BE A TUPLE OF LENGTH ≥ 1 (ALTHOUGH A LENGTH OF 1
WOULD NOT NORMALLY BE USED, AS OFSET WOULD THEN BE CALLED). */

IF TYPE(F) NE. SET THEN
  ERRTYPE(≠F(X1, ..., XN) (OFN), F IS NOT A SET!≠, F);;
IF TYPE(X) NE. TUPLE  OR.  X EQ. NULLTUPLE THEN
  ERRMSG(≠COMPILER ERROR.  OFN(F, X) CALLED WITH X NOT A ≠
     + ≠TUPLE OF LENGTH ≥ 1.≠);;

S = OFAN(F, X);
IF NMEMBS(S) EQ. 1 THEN RETURN ARBSET(S);
ELSE RETURN UNDEFWD;;
END OFN;
```

```
/*                              CFA                                    */

DEFINEF OFA(F, X);

/* THIS ROUTINE EVALUATES F≤X≥.  F MUST BE A SET.  X MAY BE
ANYTHING EXCEPT UNDEFINED.
     THE OPERATION OF THIS RCUTINE IS SIMILAR TO THAT OF
OFSET, ABOVE, EXCEPT THAT A SET OF RESULTS IS BUILT UP. */

IF TYPE(F) NE. SET THEN
  ERRTYPE(≠F≤X≥ (OFA), F IS NOT A SET:≠, F);;

IF X EQ. UNDEFWD THEN
  ERRTYPE(≠F≤X≥ (OFA), X IS UNDEFINED:≠, X);;

IF F EQ. NULLSET THEN RETURN NULLSET;;

S = NULLSET;          /* INITIALIZE RESULT TO NULL. */
SEARCH(X, F, M);      /* SEARCH FOR X IN F, STEPPING M. */
  IF TYPE(M) EQ. TUPLE THEN
    IF NCOMPS(M) EQ. 2 THEN
      IF EQUAL((TUP(M))(1), X) EQ. TRUE THEN
        /* FOUND A 2-TUPLE BEGINNING WITH X. */
        AUGACK((TUP(M))(2), S);     /* PUT 2ND COMP. IN S. */
        END IF EQUAL;
      END IF NCOMPS;
  ELSE  /* NOT A TUPLE.  CHECK FOR SPECIAL PAIR. */
    IF TYPE(M) EQ. SPECPAIR THEN
      IF EQUAL((VALUE(M))(1), X) EQ. TRUE THEN
        /* FOUND AN N-TUPLE BEGINNING WITH X. */
        AUNION(S, (VALUE(M))(2));  /* PUT ALL TAILS IN S. */
        END IF EQUAL;
      END IF TYPE(M);
    END IF TYPE(M);
  ENDSEARCH(M);

/* SEARCH IS COMPLETE. */

RETURN S;
END OFA;
```

```
DEFINEF OFAN(F,X);

/* THIS ROUTINE EVALUATES F≤X1, X2, ..., XN≥.  F MUST BE A SET
AND X MUST BE A TUPLE OF LENGTH ≥ 1 (ALTHOUGH A LENGTH OF 1
WOULD NOT NORMALLY BE USED, AS OFA WOULD THEN BE CALLED. */

IF TYPE(F) NE. SET THEN
  ERRTYPE(≠F≤X1,...,XN≥ (OFAN), F IS NOT A SET:≠, F);;
IF TYPE(X) NE. TUPLE  OR.  X EQ. NULLTUPLE THEN
  ERRMSG(≠COMPILER ERROR.  OFAN(F,X) CALLED WITH X NOT A ≠
       + ≠TUPLE OF LENGTH ≥ 1.≠);;

/* SET F IS SEARCHED FOR ALL TUPLES OF TWO OR MORE COMPONENTS
THAT BEGIN WITH X(1).  WHEN A PAIR IS FOUND, IT IS EXAMINED TO
SEE IF IT IS OF THE FORM <X1,<X2,X3,...,XN...>> OR
<X1,<X2,<X3,,,,,XN,,,,>>>, ETC.  IF SO, THE RESULT SET S
(WHICH IS INITIALLY NULL) IS AUGMENTED APPROPRIATELY.  WHEN A
SPECIAL PAIR IS FOUND THAT BEGINS WITH X1, ITS SECOND
COMPONENT (WHICH IS A SET) IS SEARCHED FOR A MATCH ON X2, X3,
X4, ..., XN.
     FOR REASONS OF EFFICIENCY, THIS ROUTINE AVOIDS RECURSION.
BUT IT USES THE RUN TIME STACK TO SAVE INTERMEDIATE RESULTS.*/

IF F EQ. NULLSET THEN RETURN NULLSET;;

S = NULLSET;                  /* INITIALIZE RESULT SET. */
I = 1;                        /* INITIALIZE ARGUMENT POINTER. */
CURF = F;                     /* INITIALIZE CURRENT SET FUNCTION.*/

L1: COMP = (TUP(X))(I);       /* GET CURRENT COMPONENT FROM ARG. */
IF COMP EQ. UNDEFWD THEN GO TO ERROR;;

SEARCH(COMP, CURF, M);        /* SEARCH FOR COMP IN CURF, STEP M.*/
  IF TYPE(M) EQ. TUPLE THEN
    IF NCOMPS(M) EQ. 2 THEN
      IF EQUAL((TUP(M))(1), COMP) EQ. TRUE THEN
        /* FOUND A POSSIBLE PAIR-CONTRIBUTOR TO F≤X1,,,,XN≥,
        I.E., <XI,A> OR <XI,<A,B,C>>. ETC. */

        M2 = (TUP(M))(2);
        K = 1;                /* INIT. POINTER TO COMPS OF M2. */
        (I+1 ≤ ^J ≤ NCOMPS(X))
          COMPJ = (TUP(X))(J);
          IF COMPJ EQ. UNDEFWD THEN I = J;  GO TO ERROR;;
          IF TYPE(M2) NE. TUPLE THEN CONTSEARCH(M);;
          IF K GE. NCOMPS(M2) THEN CONTSEARCH(M);;
```

```
            IF EQUAL((TUP(M2))(K), COMPJ) EQ. FALSE THEN
                                          CONTSEARCH(M);;
            /* ADVANCE IN M2, */
            IF (K+1) EQ. NCOMPS(M2) THEN M2 = (TUP(M2))(K+1);
                                        K = 1;
                                    ELSE K = K + 1;;
            END *J;

          /* M IS A PAIR-CONTRIBUTOR, AND M2 AND K INDICATE WHAT
          IT CONTRIBUTES, NAMELY, THE COMPONENTS OF M2 FROM K
          ON IF K > 1, AND M2 ITSELF IF K = 1. */

          IF K EQ. 1 THEN AUGACK(M2, S);
          ELSE AUGSIMP(TUPSPLT(M2,K+1,NCOMPS(M2)), S);;
          END IF EQUAL;
        END IF NCOMPS(M);
      ELSE
        IF TYPE(M) EQ. SPECPAIR THEN
          IF EQUAL((VAL(M))(1), COMP) EQ. TRUE THEN
            /* FOUND A POSSIBLE N-TUPLE CONTRIBUTOR TO
            F≤X1,...,XN≥, I.E., <XI,A,B,...> OR <XI,A,<B,...>>,
            ETC. */

            SETOFTAILS = (VAL(M))(2);
            IF I EQ. NCOMPS(X) THEN AUNION(S, SETOFTAILS);
            ELSE  /* SEARCH DEEPER. */
              PUSH(<CURF, F>);
              I = I + 1;
              CURF = SETOFTAILS;
              GO TO L1;
              END IF I;
          END IF EQUAL;
        END IF TYPE(M) EQ. SPECPAIR;
      END IF TYPE(M) EQ. TUPLE;
L2;   ENDSEARCH(M);

    /* NOW RESUME SEARCHING WHERE WE LEFT OFF ABOVE. */

    IF STACK NE. NULT. THEN
      POP(<CURF, M>);
      I = I - 1;
      COMP = (TUP(X))(I);
      GO TO L2;
    ELSE RETURN S;;

ERROR; ERRTYPE(≠F≤X1,...,XN≥ (OFAN), XI IS UNDEFINED FOR I =≠, I);

    END OFAN;
```

```
/*                              OFB                                    */

DEFINEF OFB(F, S);

/* THIS ROUTINE EVALUATES F(S).  F MUST BE A SET.  S MUST BE
A SET, TUPLE, CHARACTER STRING, OR BOOLEAN STRING.  */

IF TYPE(F) NE. SET THEN
  ERRTYPE(#F(S) (OFB), F IS NOT A SET:#, F);;
IF TYPE(S) EQ. SET   THEN RETURN OFBSET(F, S);;
IF TYPE(S) EQ. TUPLE THEN RETURN OFBTUP(F, S);;
IF TYPE(S) EQ. CHAR  THEN RETURN OFBCHAR(F, S);;
IF TYPE(S) EQ. BOOL  THEN RETURN OFBBOOL(F, S);;

ERRTYPE(#F(S) (OFB), S IS:#, S);;
END OFB;



/*                            OFBBOOL                                  */

DEFINEF OFBBOOL(F, S);

/* THIS ROUTINE EVALUATES F(S) FOR F A SET AND S A BOOLEAN
STRING.  IT IS CODED FROM THE DEFINITION F(S) = (+: B+S)F(B).
THE ROUTINE IS CODED IN A SIMPLE BUT INEFFICIENT WAY FOR
COMPACTNESS (THIS IS A LOW-USE ROUTINE).  NOTE THAT THE RESULT
MAY BE AN INTEGER, REAL, BOOLEAN STRING, CHARACTER STRING,
TUPLE, OR SET (ANYTHING FOR WHICH + IS A VALID OPERATOR).  */

IF S EQ. NULLBSTR THEN RETURN UNDEFWD;;
R = OFSET(F, OFBSTR(S, <INT, 1>));       /* R = F(S(1)).  */
(2 ≤ ∨I ≤ #VALUE(S))
  R = PLUS(R, OFSET(F, OFBSTR(S, <INT, I>)));
     /* R = R + F(S(I)).  */
  END ∨I;
RETURN R;
END OFBBOOL;
```

```
/*                          OFBCHAR                              */

DEFINEF OFBCHAR(F, S);

/* THIS ROUTINE EVALUATES F[S] FOR F A SET AND S A CHARACTER
STRING.  IT IS CODED FROM THE DEFINITION F[S] = [+: C+S]F(C).
THE ROUTINE IS CODED IN A SIMPLE BUT INEFFICIENT WAY FOR
COMPACTNESS (THIS IS A LOW-USE ROUTINE).  NOTE THAT THE RESULT
MAY BE AN INTEGER, REAL, BOOLEAN STRING, CHARACTER STRING,
TUPLE, OR RET /ANYTHING FOR WHICH + IS A VALID OPERATOR). */

IF S EQ. NULLCSTR THEN RETURN UNDEFWD;;
R = OFSET(F, OFCSTR(S, <INT, 1>));      /* R = F(S(1)), */
(2 ≤ ~I ≤ #VALUE(S))
  R = PLUS(R, OFSET(F, OFCSTR(S, <INT, I>)));
     /* R = R + F(S(I)). */
  END ~I;
RETURN R;
END OFBCHAR;



/*                          OFBTUP                               */

DEFINEF OFBTUP(F, T);

/* THIS ROUTINE EVALUATES F[T] FOR F A SET AND T A TUPLE.  IT
IS CODED FROM THE DEFINITION F[T] = [+: C+T]<F(T)> (SEE
NEWSLETTER 44), EXCEPT THAT UNDEFINED COMPONENTS IN T ARE
NOT SKIPPED OVER (AS THE DEFINITION IMPLIES). */

IF T EQ. NULLTUPLE THEN RETURN UNDEFWD;;
K = 0;                      /* K WILL KEEP TRACK OF THE HIGHEST
                               DEFINED COMPONENT IN THE RESULT. */
R = <TUPLE, <0, NCOMPALLO(NCOMPS(T)), NULT.>>;
                            /* OVERSIZED NULL TUPLE. */
(1 ≤ ~I ≤ NCOMPS(T))
  C = (TUP(T))(I);
  IF C EQ. UNDEFWD THEN (TUP(R))(I) = UNDEFWD;
                   ELSE (TUP(R))(I) = OFSET(F, C);
                        K = I;  END IF;
  END ~I;
IF K EQ. 0 THEN RETURN NULLTUPLE;;
NCOMPS(R) = K;
RETURN R;
END OFBTUP;
```

```
/*                              OFBSET                                  */

DEFINEF OFBSET(F, S);

/* THIS ROUTINE EVALUATES F[S] FOR F AND S SETS.  IT IS CODED
FROM F[S] = IF S EQ. NL. THEN NL. ELSE [+/ X→S]F≤X≥. */

IF S EQ. NULLSET THEN RETURN NULLSET;;
C = 0;                          /* CONTROL ITEM FOR ITERATION. */
R = NULLSET;
X = NEXTMEM(C, S);
(WHILE X NE. UNDEFWD)
  AUNION(R, OFA(F, X));         /* R = R + F≤X≥. */
  X = NEXTMEM(C, S);
  END WHILE;
RETURN R;
END OFBSET;


/*                              OFBN                                    */


DEFINEF OFBN(F, X);

/* THIS ROUTINE EVALUATES F[X1, X2, ..., XN].  F MUST BE A
SET AND X MUST BE A TUPLE OF LENGTH ≥ 1 (ALTHOUGH A LENGTH OF
1 WOULD NOT NORMALLY BE USED, AS OFB WOULD THEN BE CALLED).
     THIS ROUTINE IS CODED FROM THE RELATIONS F[X,Y] =
(F[X])[Y], ETC.  IT REQUIRES ↓X EVALUATIONS OF OFBSET, AND
HENCE ↓X(1) + ↓X(2) + ... + ↓X(N) EVALUATIONS OF AUNION, OFA,
AND NEXTMEM.  EVALUATION STOPS IF AN INTERMEDIATE RESULT IS
NULL, AND THE REMAINING ARGUMENTS ARE NOT CHECKED. */

IF TYPE(F) NE. SET THEN
  ERRTYPE(≠F[X1,...,XN] (OFBN), F IS NOT A SET:≠, F);;
IF TYPE(X) NE. TUPLE  OR.  X EQ. NULLTUPLE THEN
  ERRMSG(≠COMPILER ERROR,  OFBN(F,X) CALLED WITH X NOT A ≠
       + ≠TUPLE OF LENGTH ≥ 1.≠);;

R = F;                          /* INITIALIZE RESULT SET. */
(1 ≤ ~I ≤ NCOMPS(X))
  IF R EQ. NULLSET THEN QUIT ~I;;
  C = (TUP(X))(I);              /* GET NEXT ARGUMENT. */
  IF TYPE(C) NE. SET THEN
    ERRTYPE(≠F[X1,...,XN] (OFBN), ARGUMENT NOT A SET:≠, C);;
  R = OFBSET(R, C);             /* R = R[X(I)]. */
  END ~I;
RETURN R;
END OFBN;
```

6.15 FUNCTIONAL APPLICATION, STORAGE

THE ROUTINES IN THIS SECTION IMPLEMENT SETL FUNCTIONAL APPLICATION FOR LEFT HAND SIDE EXPRESSIONS. ROUTINE NAMES ARE:

SOF	F(X), MAIN ROLTINE (≠STORAGE OF≠)
SOFBSTR	F(X), F A BOOLEAN STRING
SOFCSTR	F(X), F A CHARACTER STRING
SOFTUPL	F(X), F A TUPLE
SOFSET	F(X), F A SET
SOFN	F(X1, ..., XN)
SOFA	F≤X≥
SOFAN	F≤X1, ..., XN≥
SOFB	F[X]
SOFBN	F[X1, ..., XN]
DIMFN	CHANGES F SO THAT F≤X1,...,XN≥ IS NULL

```
DEFINE SOF(F, X, R);

/* THIS IS THE MAIN ROUTINE FOR THE ASSIGNMENT F(X) = R, FOR
X A SINGLE ITEM.  THE SUBROUTINE UPDATES F BY REPLACING
(OR ADDING) ELEMENT X WITH R. */

IF TYPE(F) EQ. SET   THEN SOFSET(F, X, R);   RETURN;;
IF TYPE(F) EQ. TUPLE THEN SOFTUPL(F, X, R);  RETURN;;
IF TYPE(F) EQ. CHAR  THEN SOFCSTR(F, X, R);  RETURN;;
IF TYPE(F) EQ. BOOL  THEN SOFBSTR(F, X, R);  RETURN;;
IF TYPE(F) EQ. FUN  THEN
  ERRMSG(≠COMPILER ERROR OR ATTEMPT TO USE A FEATURE NOT YET ≠
       + ≠IMPLEMENTED.  THE ROUTINE FOR F(X) (SOF) WAS CALLED≠
       + ≠ WITH F A PROCEDURE.  THE CALL MUST BE HANDLED BY ≠
       + ≠COMPILER GENERATED CODE, NOT BY THE SRTL OF ≠
       + ≠ROUTINE.≠);
  RETURN;;
ERRTYPE(≠F(X) (SOF), F IS NEITHER A SET, TUPLE, NOR STRING, ≠
       + ≠ F IS: ≠, F);
END SOF;
```

```
/*                          SOFBSTR                                  */

DEFINE SOFBSTR(F, X, R);

/* THIS ROUTINE IMPLEMENTS F(X) = R FOR F A BOOLEAN STRING,
BIT X OF F IS REPLACED BY R. */

I = OKINDEX(X);              /* VERIFY REASONABLENESS OF X, */
IF I NE. 0 THEN
  IF R EQ. TRUE  OR.  R EQ. FALSE THEN
    (VALUE(F))(I) = (VALUE(R))(1);
    RETURN;
  ELSE
    ERRTYPE(#F(X) = R (SOFBSTR), R IS NOT A BOOLEAN STRING #
          + #OF LENGTH ONE!#. R);
    END IF;
ELSE
  ERRTYPE(#F(X) = R (SOFBSTR), OR UNREASONABLE VALUE OF INDEX#
        #, X = #, X);
  END IF I NE. 0;
END SOFBSTR;


/*                          SOFCSTR                                  */

DEFINE SOFCSTR(F, X, R);

/* THIS ROUTINE IMPLEMENTS F(X) = R FOR F A CHARACTER STRING.
CHARACTER X OF F IS REPLACED BY R. */

I = OKINDEX(X);
IF I NE. 0 THEN
  IF TYPE(F) EQ. CHAR  AND.  #VALUE(R) EQ. 1 THEN
    (VALUE(F))(I) = (VALUE(R))(1);
    RETURN;
  ELSE
    ERRTYPE(#F(X) = R (SOFCSTR), R IS NOT A CHARACTER STRING #
          + #OF LENGTH ONE!#, R);
    END IF;
ELSE
  ERRTYPE(#F(X) = R (SOFCSTR), OR UNREASONABLE VALUE OF INDEX#
        + #, X = #, X);
  END IF I NE. 0;
END SOFCSTR;
```

```
/*                          SOFTUPL                          */

DEFINE SOFTUPL(F, X, R);

/* THIS ROUTINE IMPLEMENTS F(X) = R FOR F A TUPLE.  COMPONENT
X OF F IS REPLACED BY R. */

I = OKINDEX(X);
IF I NE. 0 THEN
  (TUP(F))(I) = R;
  /* IF R = UNDEFWD AND I = NCOMPS(F), THE TUPLE MUST BE
  CONTRACTED. */

  (WHILE(TUP(F))(NCOMPS(F)) EQ. UNDEFWD)
    NCOMPS(F) = NCOMPS(F) - 1;
    IF NCOMPS(F) EQ. 0 THEN F = NULLTUPLE;  QUIT WHILE;;
    END WHILE;
  RETURN;

ELSE
  ERRTYPE(≠F(X) = R (SOFTUPL), OR UNREASONABLE VALUE OF INDEX≠
        ≠, X = ≠, X);
  END IF I NE. 0;
END SOFTUPL;



/*                          OKINDEX                          */

DEFINEF OKINDEX(X);

/* THIS ROUTINE RETURNS THE VALUE OF X IF X IS AN INTEGER OF
REASONABLE VALUE FOR AN INDEX INTO A TUPLE OR STRING, AND 0
OTHERWISE. */

IF TYPE(X) EQ. INT THEN
  VX = VALUE(X);
  IF VX GT. 0  AND.  VX LT. 1000000 THEN RETURN VX;;;
RETURN 0;
END OKINDEX;
```

```
/*                          SOFSET                                  */

DEFINE SOFSET(F, X, R);

/* THIS ROUTINE IMPLEMENTS F(X) = R FOR F A SET.  IT IS CODED
FROM THE EQUIVALENT ASSIGNMENT:

     F = (F LESF. X) WITH. <X,R>.    */

IF X EQ. UNDEFWD THEN
  ERRTYPE(≠F(X) = R (SOFSET), X IS UNDEFINED:≠, X);;

DIMFAOK(F, X);              /* F = F LESF. X. */
IF R NE. UNDEFWD THEN
  AUGSIMP(<TUPLE,<2,2,<X,R>>>, F); /* F = F WITH. <X,R>. */
  END IF;
RETURN;
END SOFSET;



/*                          SOFN                                    */

DEFINE SOFN(F, X, R);

/* THIS ROUTINE IMPLEMENTS F(X1,X2,...,XN) = R.  F MUST BE
A SET, AND ALL THE XI MUST BE DEFINED.  THE ROUTINE IS CODED
FROM THE EQUIVALENT:

     DIMFN(F, X);
     F = F WITH. <X1,X2,...,XN,R>.

WHERE DIMFN IS SIMILAR TO DIMF (SETL LESF); SEE BELOW. */

IF TYPE(F) NE. SET THEN
  ERRTYPE(≠F(X1,...,XN) = R (SOFN), F IS NOT A SET:≠, F);;

IF TYPE(X) NE. TUPLE  OR,  X EQ. NULLTUPLE THEN
  ERRMSG(≠COMPILER ERROR,  SOFN(F,X,R) CALLED WITH X NOT A ≠
       + ≠TUPLE OF LENGTH ≥ 1.≠);;

DIMFN(F, X);
IF R NE. UNDEFWD THEN
  TUPADD1(X, R);
  AUGTUP(X, F);;
RETURN;
END SOFN;
```

```
/*                              SOFA                                    */

      DEFINE SOFA(F, X, R);

      /* THIS ROUTINE IMPLEMENTS F≤X≥ = R, WHERE X IS A SINGLE ITEM.
      F AND R MUST BE SETS, AND X MUST BE DEFINED.  THE ROUTINE IS
      CODED FROM THE EQUIVALENT:

            F = F LESF. X;
            (~M → R)  F = F WITH. <X,M>;;   */

      IF TYPE(F) NE. SET THEN
        ERRTYPE(≠F≤X≥ = R (SOFA), F IS NOT A SET:≠, F);;

      IF X EQ. UNDEFWD THEN
        ERRTYPE(≠F≤X≥ = R (SOFA), X IS UNDEFINED:≠, X);;

      IF TYPE(R) NE. SET THEN
        ERRTYPE(≠F≤X≥ = R (SOFA), R IS NOT A SET:≠, R);;

      DIMFAOK(F, X);                          /* F = F LESF. X. */

      C = 0;
LOOP: M = NEXTMEM(C, R);
      IF M EQ. UNDEFWD THEN GO TO OUT;;
      AUGSIMP(<TUPLE,<2,2,<X,M>>>, F);  /* F = F WITH. <X,M>. */
      GO TO LOOP;

OUT:  RETURN;
      END SOFA;
```

```
/*                          SOFAN                                  */

      DEFINE SOFAN(F, X, R);

      /* THIS ROUTINE IMPLEMENTS F≤X1,X2,...,XN> = R.  F AND R MUST
      BE SETS AND ALL THE XI MUST BE DEFINED.  THE ROUTINE IS CODED
      FROM THE EQUIVALENT:

           DIMFN(F, X);
           (∀M → R)  F = F WITH. <X1,X2,...,XN,M>;;

      THIS IS NOT VERY EFFICIENT, BUT IT IS EXPECTED TO BE A LOW-USE
      ROUTINE. */

      IF TYPE(F) NE. SET THEN
        ERRTYPE(≠F≤X1,...,XN≥ = R (SOFAN), F IS NOT A SET:≠, F);;

      IF TYPE(X) NE. TUPLE  OR.  X EQ. NULLTUPLE THEN
        ERRMSG(≠COMPILER ERROR,  SOFAN(F,X,R) CALLED WITH X NOT A ≠
             + ≠TUPLE OF LENGTH ≥ 1.≠);;

      IF (TYPE(R) NE. SET THEN
        ERRTYPE(≠F≤X1,...,XN≥ = R (SOFAN), R IS NOT A SET:≠, R);;

      DIMFN(F, X);
      N = NCOMPS(X) + 1;
      IF N GT. SPACE(X) THEN VALUE(X) = <N, N, X>;
                        ELSE NCOMPS(X) = N;  END IF;
      C = 0;
LOOP: M = NEXTMEM(C, R);
      IF M EQ. UNDEFWD THEN GO TO OUT;;
      (TUP(X))(N) = M;
      AUGTUP(X, F);      /* F = F WITH. <X1,...,XN,M>. */
      GO TO LOOP;

 OUT: RETURN;
      END SOFAN;
```

```
/*                          SCFB                                    */

DEFINE SOFB(F, S, R);

/* THIS ROUTINE IMPLEMENTS F[S] = R, WHERE S IS A SINGLE ITEM.
F, S, AND R MUST ALL BE SETS.  IT IS CODED FROM:

     (~X → S) F = F LESF. X;;
     (~X → S, M → R) F = F WITH. <X,M>;;  */

IF TYPE(F) NE. SET THEN
  ERRTYPE(≠F[S] = R (SOFB), F IS NOT A SET:≠, F);;

IF TYPE(S) NE. SET THEN
  ERRTYPE(≠F[S] = R (SOFB), S IS NOT A SET:≠, S);;

IF TYPE(R) NE. SET THEN
  ERRTYPE(≠F[S] = R (SOFB), R IS NOT A SET:≠, R);;

C = 0;
LOOP:  X = NEXTMEM(C, S);
       IF X EQ. UNDEFWD THEN GO TO OUT;;
       DIMFAOK(X, F);          /* F = F LESF. X. */
       GO TO LOOP;

 OUT:  CS = 0;
LOOP1:  X = NEXTMEM(CS, S);
        IF X EQ. UNDEFWD THEN GO TO OUT1;;
        CR = 0;
LOOP2:    M = NEXTMEM(CR, R);
          IF M EQ. UNDEFWD THEN GO TO OUT2;;
          AUGSIMP(<TUPLE, <2,2,<X,M>>>, F); /* F = F WITH. <X,M>, */
          GO TO LOOP2;
 OUT2:  GO TO LOOP1;

 OUT1:  RETURN;
        END SOFB;
```

```
DEFINE SCFBN(F, S, R);

/* THIS ROUTINE IMPLEMENTS F[S(1), S(2), ..., S(N)] = R.  F,
S(1), ..., S(N), AND R MUST ALL BE SETS.  IT IS CODED FROM
(USING SETL LOOSELY):

      (∀X(1)∈S(1), X(2)∈S(2), ..., X(N)∈S(N)) DIMFN(X, F);
      (∀X(1)∈S(1), X(2)∈S(2), ..., X(N)∈S(N), M∈R)
        F = F WITH, <X(1), X(2), ..., X(N), M>;;  */

IF TYPE(F) NE. SET THEN
  ERRTYPE(≠F[S1,...,SN] = R (SCFBN), F IS NOT A SET;≠, F);;

IF TYPE(S) NE. TUPLE  OR.  S EQ. NULLTUPLE THEN
  ERRMSG(≠COMPILER ERROR.  SCFBN(F, S, R) CALLED WITH S ≠
       + ≠NOT A TUPLE OF LENGTH ≥ 1.≠);;

IF TYPE(R) NE. SET THEN
  ERRTYPE(≠F[S1,...,SN] = R (SCFBN), R IS NOT A SET;≠, R);;

N = NCOMPS(S);              /* S IS A TUPLE OF SETS. */
I = 1;
C = <TUPLE, <N+1, N+1, <0>>>;       /* ROOM FOR N+1 COMPONENTS
                                       WITH C(1) = 0. */
X = <TUPLE, <N, N+1, NULT.>>;
(WHILE I GT. 0)
  (TUP(X))(I) = NEXTMEM((TUP(C))(I), (TUP(S))(I));
  IF (TUP(X))(I) EQ. UNDEFWD THEN I = I - 1;
                                  CONTINUE WHILE;;
  IF I LT. N THEN I = I + 1;
                  (TUP(C))(I) = 0;
                  CONTINUE WHILE;;

  /* I = N AND X(I) IS DEFINED.  REMOVE FROM F ALL TUPLES THAT
  CORRESPOND TO X. */

  DIMFN(X, F);
  END WHILE;

IF R EQ. NULLSET THEN RETURN;;      /* (INCLUDED FOR EFFICIENCY
                                       ONLY). */
```

```
/* NOW AUGMENT F WITH ALL (N+1)-TUPLES OF THE FORM
<X1,X2,...,XN,M>, WX1→S1, X2→S2, ..., XN→SN, M→R.  NOTE THAT
C AND X ALREADY HAVE ROOM FOR N+1 COMPONENTS. */

TUPADD1(S, R);
I = 1;
(TUP(C))(1) = 0;
NCOMPS(X) = N+1;

(WHILE I GT. 0)
  (TUP(X))(I) = NEXTMEM((TUP(C))(I), (TUP(S))(I));
  IF (TUP(X))(I) EQ. UNDEFWD THEN I = I - 1;
                                  CONTINUE WHILE;;
  IF I LE. N THEN I = I + 1;
                  (TUP(C))(I) = 0;
                  CONTINUE WHILE;;

  /* I = N+1 AND X(N+1) IS DEFINED.  ADD THE (N+1)-TUPLE
  <X1,X2,...,XN,M> TO F. */

  AUGTUP(X, F);
  END WHILE;

RETURN;
END SOFBN;
```

```
/*                          DIMFN                                  *

DEFINE DIMFN(F, X);

/* FOR THIS ROUTINE, F IS A SET AND X IS AN N-TUPLE, N ≥ 1,
THE ROUTINE REMOVES FROM SET F ALL TUPLES OF ANY OF THE FORMS

                    <X1, X2, ..., XN, ...>
                    <X1, X2, ..., <XN,...>>
                    ...
                    <X1, <X2, ..., <XN,...>>>

HOWEVER, THE TUPLE <X1, X2, ..., XN> IS LEFT IN F, SO AS NOT
TO CHANGE THE VALUE OF F(X1, X2, ..., X(N-1)). */

IF F EQ. NULLSET THEN RETURN;;

I = 1;                      /* INITIALIZE INDEX OF X. */
S = F;                      /* INITIALIZE CURRENT SET. */

L1: NBRDELETED = 0;         /* NUMBER OF MEMBERS DELETED FROM S*/
COMP = (TUP(X))(I);         /* CURRENT COMPONENT OF X. */
HSH = HASH(COMP);
ENTRY = HSH//HTSIZE(S) + 1;
M = (HASHTABLE(S))(ENTRY);
PREV = OM.;
(WHILE M NE. OM.  DOING M = NEXTM(M);)
  IF TYPE(M) EQ. TUPLE THEN
    IF NCOMPS(M) EQ. 2 THEN
      IF EQUAL((TUP(M))(1), COMP) EQ. TRUE THEN
        /* FOUND A MATCH ON A PAIR, I.E., M = <XI,A> OR
        <XI,<A,B,C>>, ETC.  SEARCH DEEPER INTO M, IF THERE ARE
        MORE COMPONENTS OF X. */

        M2 = (TUP(M))(2);
        K = 1;              /* SET INDEX FOR M2. */
        (I+1 ≤ ~J ≤ NCOMPS(X))
          IF TYPE(M2) NE. TUPLE THEN GO TO L2;;
          IF K GE. NCOMPS(M2) THEN GO TO L2;;
          COMPJ = (TUP(X))(J);
          IF EQUAL((TUP(M2))(K),COMPJ)EQ.FALSE THEN GO TO L2;;
          /* ADVANCE IN M2. */
          IF (K+1) EQ. NCOMPS(M2) THEN M2 = (TUP(M2))(K+1);
                                        K = 1;
                                  ELSE K = K + 1;;
          END ~J;
```

```
            /* M MATCHES THE ARGUMENT X.  DELETE IT FROM S.  NOTE:
            DIMSIMP MAY NOT BE USED, BECAUSE WE CANNOT CONTRACT S
            WHILE WE ARE ITERATING OVER IT. */

            IF PREV NE. OM. THEN NEXTM(PREV) = NEXTM(M);
                           ELSE (HASHTABLE(S))(ENTRY)=NEXTM(M);;
            NBRDELETED = NBRDELETED + 1;
            LOAD(S) = LOAD(S) - 1;
            HCODE(S) = ((HCODE(S) AS. BSTRING) //
                        (HSH AS. BSTRING)) AS. INT;
            CONTINUE WHILE(M);
            END IF EQUAL;
          END IF NCOMPS;
        ELSE
          IF TYPE(M) EQ. SPECPAIR THEN
            IF EQUAL((VAL(M))(1), COMP) EQ. TRUE THEN

            /* FOUND A MATCH ON AN N-TUPLE, N ≥ 3, I.E.,
            <X1,A,B,...> OR <X1,A,<B,...>>, ETC. */

            SETOFTAILS = (VALUE(M))(2);
            IF I EQ. NCOMPS(X) THEN
              /* DELETE THE SPECIAL PAIR M FROM S. */
              IF PREV NE. OM. THEN NEXTM(PREV) = NEXTM(M);
              ELSE (HASHTABLE(S))(ENTRY) = NEXTM(M);;
              NBRDELETED = NBRDELETED + NMEMBS(SETOFTAILS);
              LOAD(S) = LOAD(S) - 1;
              HCODE(S) = ((HCODE(S) AS. BSTRING.) //
                          (HSH AS. BSTRING.)) AS. INT.;
              CONTINUE WHILE(M);
            ELSE /* I < NCOMPS(X).  SEARCH DEEPER. */
              PUSH(<S, ENTRY, M, NBRDELETED, HSH>);
              I = I + 1;
              S = SETOFTAILS;
              GO TO L1;
              END IF I;       /* (NEVER FALL THROUGH HERE). */
            END IF EQUAL;
          END IF TYPE(M) EQ. SPECPAIR;
        END IF TYPE(M) EQ. TUPLE;

L2:   PREV = M;

L3:   END WHILE(M);

    /* THROUGH ITERATING OVER S.  CONTRACT ITS HASH TABLE IF
    NECESSARY. */

    NMEMBS(S) = NMEMBS(S) - NBRDELETED;
    IF NMEMBS(S) EQ. 0 THEN S = NULLSET;
    ELSE
```

```
    (WHILE TOOSPARSE(S)  AND.  HTSIZE(S) GT. MINHTSIZE)
      CONTRCT(S);
      END WHILE;
    END IF;

/* NOW POP THE STACK AND CONTINUE ITERATING AT THE HIGHER
LEVEL (IF THE STACK IS NON-EMPTY).  HOWEVER, IF THE SET S
ABOVE BECAME NULL, IT IS NECESSARY TO DELETE THE SPECIAL PAIR
M AT THE TOP OF THE STACK FROM THE SET S AT THE TOP OF THE
STACK. */

IF STACK NE. NULT. THEN
    TEMP = (S EQ. NULLSET);
    POP(<S, ENTRY, M, NBRDELETED, HSH>);
    I = I - 1;
    COMP = (TUP(X))(I);
    IF TEMP THEN
      /* DELETE THE SPECIAL PAIR M FROM S. */
      IF PREV NE. OM. THEN NEXTM(PREV) = NEXTM(M);
                      ELSE (HASHTABLE(S))(ENTRY) = NEXTM(M);;
      LOAD(S) = LOAD(S) - 1;
      HCODE(S) = ((HCODE(S) AS. BSTRING.) //
                  (HSH AS. BSTRING.)) AS. INT.;
      END IF TEMP;
    GO TO L3;
ELSE
    /* STACK IS EMPTY.  WE WERE ITERATING OVER THE TOP LEVEL. */
    F = S;                    /* SET RESULT. */
    RETURN;
    ENDIF;

END DIMFN;
```

6.16 COPY RCUTINES

```
DEFINEF COPY(R);     /* ENTRY POINT FOR THE SETL COPY(R). */

/* THIS FUNCTION RETURNS A CCPY OF ITS ARGUMENT.  THE OBJECT
IS COPIED TO ITS FULL DEPTH.  AN EXCEPTION IS MADE OF THE NULL
TUPLE AND NULL SET:  THEY ARE NOT COPIED.  THIS IS BECAUSE
ROUTINES THAT NORMALLY MODIFY TUPLES AND SETS MAKE AN
EXCEPTION OF THE NULL TUPLE AND NULL SET, AND NEVER MODIFY
THEM.  HENCE WE AVOID HAVING SEVERAL COPIES OF THESE OBJECTS,
WHICH WOULD BE A WASTE OF SPACE.
     THE COPY ROUTINE IN THE LITTLE VERSION OF THE RUN TIME
LIBRARY IS LESS SENSITIVE TO OBJECT TYPES THAN THE ONE GIVEN
BELOW.  IT WCRKS WITH OBJECTS AT THE ≠GARBAGE COLLECTION
LEVEL≠, WITH THE EXCEPTION OF CHECKS FOR THE NULL TUPLE AND
SET.  HENCE THE CODE BELOW DCES NOT ACCURATELY REFLECT THAT IN
THE LITTLE RUN TIME LIBRARY; IT SHOWS WHAT IS TO BE DONE BUT
NOT HOW TO DC IT.  */

C = <TYPE(R)>;       /* INITIALIZE RESULT. */
IF NEXTM(R) NE. OM. THEN
  /* COPY THE SET-LIST CONNECTED TO R. */
  NEXTM(C) = COPY(NEXTM(R));;

/* NOW COPY THE VALUE FIELD CF R. */

FLOW                 TYPE(R) EC. TUPLES
  R(1:2) EQ. NULLTUPLES          TYPE(R) EQ. SETS
  NULLT.       COPYTUP,  R(1:2)EQ.NULLSETS TYPE(R)EQ.SPECPAIRS
                         NULLS,   COPYSET, COPYSP,   COPYATOM;

  NULLT:     VALUE(C) = VALUE(NULLTUPLE);      /* (NO COPY). */
  COPYTUP:   VALUE(C) = <NCOMPS(R), SPACE(R), NULT.>;
             (1≤∨I≤NCOMPS(R)) (TUP(C))(I) = COPY((TUP(R))(I));;
  NULLS:     VALUE(C) = VALUE(NULLSET);        /* (NO COPY). */
  COPYSET:   VALUE(C) = <NMEMES(R), LOAD(R), HCODE(R),
                         HTSIZE(R), NULT.>;
             (1 ≤ ∨I ≤ HTSIZE(R))
               (HASHTABLE(C))(I) = COPY((HASHTABLE(R))(I));;
  COPYSP:    (1 ≤ ∨I ≤ 2) (VALUE(C))(I) = COPY((VALUE(R))(I));;
  COPYATOM:  VALUE(C) = VALUE(R);
             END FLOW;

RETURN C;
END COPY;
```

```
/*                          COPY1                              */

DEFINEF COPY1(R);

/* THIS FUNCTION COPIES R ONE LEVEL DEEP.  IT EXISTS IN THE
LITTLE VERSION OF THE RUN TIME LIBRARY BECAUSE IT PROVIDES
A FASTER WAY TO COPY LONG CHARACTER STRINGS, LONG INTEGERS,
ETC., AND ALSO BECAUSE IT MAY BE AN ADEQUATE TYPE OF COPY FOR
TUPLES AND SETS IN CERTAIN SITUATIONS.
     AS IN THE CASE OF THE COPY ROUTINE, THIS DOES NOT
ACCURATELY REFLECT THE CODE OF THE LITTLE VERSION.  */

/* THE LITTLE VERSION OF THIS ROUTINE WILL COPY THE FIRST ITEM
IN A SET-LIST.  HOWEVER, THIS FEATURE IS NEVER USED, BECAUSE
NEXTM(R) IS ALWAYS UNDEFINED WHEN THIS ROUTINE IS USED, AND
FOR SIMPLICITY IT IS NOT SHOWN HERE.  HOWEVER, TO PREVENT
INCOMPATIBLE USE OF THIS ROUTINE, WE CHECK TO ASSURE THAT
NEXTM(R) IS ALWAYS UNDEFINED.  */

IF NEXTM(R) NE. OM. THEN
  ERRMSG(≠COPY1 CALLED WITH AN OBJECT IN A SET-LIST.≠);

C ≔ <TYPE(R)>;        INITIALIZE RESULT. */

/* NOW COPY THE VALUE FIELD OF R. */

FLOW                  TYPE(R) EQ. TUPLE$
  R(1:2) EQ. NULLTUPLE$              TYPE(R) EQ. SET$
  NULLT,       COPYTUP,  R(1:2)EQ.NULLSET$ TYPE(R)EQ.SPECPAIR$
                         NULLS,    COPYSET, COPYSP,    COPYATOM;

  NULLT:     VALUE(C) ≔ VALUE(NULLTUPLE);      /* (NO COPY). */
  COPYTUP:   VALUE(C) ≔ <NCOMPS(R), SPACE(R), NULT.>;
             (1≤∀I≤NCOMPS(R)) (TUP(C))(I) ≔ (TUP(R))(I);
  NULLS:     VALUE(C) ≔ VALUE(NULLSET);        /* (NO COPY). */
  COPYSET:   VALUE(C) ≔ <NMEMES(R), LOAD(R), HCODE(R),
                         HTSIZE(R), NULT.>;
             (1 ≤ ∀I ≤ HTSIZE(R))
               (HASHTABLE(C))(I) ≔ (HASHTABLE(R))(I);;
  COPYSP:    (1 ≤ ∀I ≤ 2) (VALUE(C))(I) ≔ (VALUE(R))(I);;
  COPYATOM:  VALUE(C) ≔ VALUE(R);
             END FLOW;

RETURN C;
END COPY1;
```

6.17 HEAD AND TAIL ROUTINES

```
DEFINEF HEAD(X);     /* ENTRY POINT FOR THE SETL HD. X. */

FLOW          TYPE(X) EQ. TUPLE$
  NCOMPS(X) GE. 1$          X EQ. UNDEFWD$
  TUP1,        UND,         UNI,       ERR;

  TUP1:      RETURN (TUP(X))(1);
  UND:       RETURN UNDEFWD;
  ERR:       ERRTYPE(#HEAD(X), X IS#, X);
             END FLOW;
END HEAD;



/*                          TAIL                                  */


DEFINEF TAIL(X);     /* ENTRY POINT FOR THE SETL TL. X. */

FLOW          TYPE(X) EQ. TUPLE$
  NCOMPS(X) GE. 2$          X EQ. UNDEFWD$
  MAKETAIL,    UND,         UNI,       ERR;

  MAKETAIL: L = NCOMPS(X) - 1;
            T = <TUPLE, <L, NCOMPALLO(L), NULT.>>;
            (1 ≤ ∀I ≤ L) (TUP(T))(I) = (TUP(X))(I);;
            RETURN T;
  UND:      RETURN UNDEFWD;
  ERR:      ERRTYPE(#TAIL(X), X IS#, X);
            END FLOW;
END TAIL;
```

6.18 PLUS ROUTINES

```
DEFINE PLUS(X, Y);          /* ENTRY POINT FOR THE SETL X + Y. */

/* THIS SUBROUTINE SETS X = X + Y FOR ALL THE SETL MEANINGS OF
THE PLUS SIGN. */

/* NOTE: THIS ROUTINE AND SOME THAT FOLLOW USE AN UNORTHODOX
LAYOUT OF THE FLOW STATEMENT, BUT IT IS SYNTACTICALLY CORRECT
AND ITS MEANING SHOULD BE CLEAR. */

IF TYPE(X) NE. TYPE(Y) THEN
  ERRMIX(#PLUS(X,Y).  X, Y =#, X, Y);;

FLOW TYPE(X) EQ. INTS   (VALUE(X) = VALUE(X) + VALUE(Y);),
     TYPE(X) EQ. REALS  (VALUE(X) = VALUE(X) + VALUE(Y);),
     TYPE(X) EQ. BOOLS  (VALUE(X) = VALUE(X) + VALUE(Y);),
     TYPE(X) EQ. CHARS  (VALUE(X) = VALUE(X) + VALUE(Y);),
     TYPE(X) EQ. TUPLES (CONCATT(X, Y);),
     TYPE(X) EQ. SETS   (AUNION(X, Y);),
     (ERRTYPE(#PLUS(X,Y).  X IS#, X););
     END FLOW;

RETURN;
END PLUS;
```

```
/*                    CONCATT                    */

DEFINE CONCATT(T1, T2);

/* THIS SUBRCUTINE CONCATENATES T1 AND T2.  THE RESULT IS IN
T1.  THE CONCATENATION IS DONE IN-PLACE IF POSSIBLE.  THE
RESULTING TUPLE MAY HAVE TRAILING UNDEFINED COMPONENTS. */

LENGTH1 = NCOMPS(T1);
LENGTH2 = NCOMPS(T2);
NEWLENGTH = LENGTH1 * LENGTH2;

IF NEWLENGTH GT. SPACE(T1) THEN
  /* INSUFFICIENT ROOM.  REALLOCATE T1. */
  R = <TUPLE, <NEWLENGTH, NCOMPALLO(NEWLENGTH), NULT.>>;
  /* MOVE COMPONENTS FROM T1 TO NEW SPACE. */
  (1 ≤ ~I ≤ LENGTH1)
    (TUP(R))(I) = (TUP(T1))(I);
    END ~I;
  T1 = R;
  END IF;

/* NOW ATTACH T2 TO THE END OF T1. */
(1 ≤ ~I ≤ LENGTH2)
  (TUP(T1))(LENGTH1 * I) = (TUP(T2))(I);
  END ~I;
NCOMPS(T1) = NEWLENGTH;
RETURN;
END CONCATT;
```

```
/*                          AUNION                              */

DEFINE AUNION(X, Y);

/* THIS IS THE #AUGMENTING UNION# ROUTINE.  X AND Y ARE
SETS, AND ON RETURN X = X + Y.  THE ROUTINE ITERATES
OVER Y, PUTTING EACH MEMBER OF Y IN X.  HENCE IT EXECUTES
FASTER IF Y IS THE SMALLER SET.
     IT IS POSSIBLE THAT THIS ROUTINE SHOULD SWITCH ARGUMENTS
WHEN THE FIRST IS SMALLER, SO THAT IT ALWAYS ITERATES OVER THE
SMALLER SET.  TO DO THIS WOULD REQUIRE FIRST COPYING Y, AS
THE ROUTINE SHOULD NEVER ALTER ITS SECOND ARGUMENT.  THE
QUESTION THEN ARISES:  HOW DEEPLY SHOULD Y BE COPIED.  ITS
SET-LISTS WOULD HAVE TO BE COPIED, BUT ITS MEMBERS WOULD NOT.
HENCE WE SHOULD HAVE A SPECIAL COPY ROUTINE THAT COPIES ONE
LEVEL DEEP (THIS WOULD BE DIFFERENT FROM COPY1, AS COPY1 DOES
NOT COPY SET-LISTS.  ALTERNATIVELY, COPY1 COULD BE
REDESIGNED).
     FOR THE PRESENT WE TAKE THE SIMPLE APPROACH OF ALWAYS
ITERATING OVER THE SECOND ARGUMENT, MODIFYING THE FIRST.
     IN ADDITION TO THE PLUS MAIN ROUTINE, THIS ROUTINE IS
CALLED BY F≤X≥ (OFA), F≤X1,..,XN≥ (OFAN), AND F{S} (OFBSET).*/

C = 0;                  /* INITIALIZE CONTROL FOR NEXTMEM. */

L1:   MY = NEXTMEM(C, Y);
      IF MY EQ. UNDEFWD THEN RETURN;;
      AUGAOK(MY, X);
      GO TO L1;

END AUNION;
```

6.19 MINUS ROUTINES

```
DEFINE MINUS(X, Y);       /* ENTRY POINT FOR THE SETL X - Y. */

/* THIS SUBROUTINE SETS X = X - Y FOR ALL THE SETL MEANINGS OF
THE MINUS SIGN. */

IF TYPE(X) NE. TYPE(Y) THEN
  ERRMIX(≠MINUS(X,Y).  X, Y =≠, X, Y);;

FLOW TYPE(X) EQ. INTS    (PMINUS(X);
                          VALUE(X) = VALUE(X) * VALUE(Y);
                          PMINUS(X);),
     TYPE(X) EQ. REALS   (VALUE(X) = VALUE(X) - VALUE(Y);),
     TYPE(X) EQ. BOOLS   (VX = VALUE(X);
                          VY = VALUE(Y);
                          VALUE(X) = IF ↓VX LE. ↓VY THEN
                                       VX AND. NOT. VY
                                     ELSE NOT.(NOT.VX OR. VY);).
     TYPE(X) EQ. SETS    (SETDIFF(X, Y);),
     (ERRTYPE(≠MINUS(X,Y).  X IS≠, X);)
     END FLOW;
RETURN;
END MINUS;


/*                          SETDIFF                              */


DEFINE SETDIFF(X, Y);

/* THIS SUBROUTINE SETS X = X - Y FOR X AND Y SETS.  IT
ITERATES OVER Y, REMOVING EACH MEMBER OF Y FROM X.  HENCE IT
EXECUTES FASTER IF Y IS THE SMALLER SET. */

C = 0;                /* INITIALIZE CONTROL FOR NEXTMEM. */

L1:  MY = NEXTMEM(C, MY);
     IF MY EQ. UNDEFWD THEN RETURN;;
     DIMAOK(MY, X);
     GO TO L1;

END SETDIFF;
```

```
/*                    PMINUS                                    *

DEFINE PMINUS(X);          /* ENTRY POINT FOR THE SETL -X. */

/* THIS SUBROUTINE CHANGES X TO -X. */

IF TYPE(X) EQ. INT  OR.  TYPE(X) EQ. REAL THEN
  VALUE(X) = -VALUE(X);
  RETURN;
  END IF;

ERRTYPE(≠PMINUS(X).  X IS≠, X);
END PMINUS;
```

6.20 MULTIPLICATION ROUTINES

```
DEFINE MULT(X, Y, R);     /* ENTRY POINT FOR THE SETL X*Y, */

/* THIS SUBRCUTINE SETS R = X*Y FOR ALL THE SETL MEANINGS OF
THE ASTERISK, */

IF TYPE(X) EQ. INT THEN
  IF TYPE(Y) EQ. INT OR. TYPE(Y) EQ. BOOL OR. TYPE(Y) EQ. CHAR
    THEN /* INTEGER MULTIPLY OR REPLICATION OF A STRING. */
    R = <TYPE(Y), VALUE(X)*VALUE(Y)>;  RETURN;
  ELSE ERRTYPE(≠MULT(X,Y), X IS AN INTEGER, Y IS≠, Y);;;

IF TYPE(X) EQ. REAL THEN
  IF TYPE(Y) EQ. REAL THEN
    R = <REAL, VALUE(X)*VALUE(Y)>;  RETURN;
  ELSE ERRTYPE(≠MULT(X,Y), X IS REAL, Y IS≠, Y);;;

IF TYPE(X) EQ. SET THEN
  IF TYPE(Y) EQ. SET THEN INTSECT(X, Y, R);  RETURN;
  ELSE ERRTYPE(≠MULT(X,Y), X IS A SET, Y IS≠, Y);;;

ERRTYPE(≠MULT(X,Y), X IS≠, X);
END MULT;
```

```
/*                          INTSECT                                  */

      DEFINE INTSECT(X, Y, R);

      /* THIS SUBRCUTINE SETS R = X*Y FOR X AND Y SETS (SET
      INTERSECTION).
           THE INTERSECTION SET IS BUILT BY ITERATING OVER THE
      SMALLER OF X AND Y, CHECKING TO SEE IF EACH MEMBER OF THE
      SMALLER SET IS ALSO IN THE LARGER SET.  IF SO, IT IS ADDED TO
      THE RESULT SET R. */

      IF NMEMBS(X) LE. NMEMBS(Y) THEN SMALLER = X;
                                      LARGER = Y;
                                 ELSE SMALLER = Y;
                                      LARGER = X;;

      R = NULLSET;          /* INITIALIZE RESULT. */
      C = 0;                /* INITIALIZE CONTROL FOR NEXTMEM. */

LOOP: M = NEXTMEM(C, SMALLER);
      IF M EQ. UNDEFWD THEN RETURN;;
      IF ELMTSET(M, LARGER) EQ. TRUE THEN AUGAOK(M, R);;
      GO TO LOOP;

      END INTSECT;
```

6.21 DIVISION ROUTINES

```
DEFINE DIVIDE(X, Y, R);  /* ENTRY POINT FOR THE SETL X/Y. */

/* THIS SUBROUTINE SETS X = X/Y FOR ALL THE SETL MEANINGS OF
THE SLASH.  IN ADDITION, IF X AND Y ARE INTEGERS, R IS SET
EQUAL TO THE REMAINDER.  THE QUOTIENT AND REMAINDER ARE
RELATED BY X = Q*Y + R, WITH SIGN(R) = SIGN(X) AND ABS(R) < Y.
FOR EXAMPLE:
                9/4  =  2 REM 1,     -9/4  = -2 REM -1,
                9/-4 = -2 REM 1,     -9/-4 =  2 REM 1.  */

IF TYPE(X) NE. TYPE(Y) THEN
  ERRMIX(#DIVIDE(X,Y),  X,Y =#, X, Y);;

FLOW TYPE(X) EQ. INTS  (R = VALUE(X)//VALUE(Y);
                        VALUE(X) = VALUE(X)/VALUE(Y);),
     TYPE(X) EQ. REALS (VALUE(X) = VALUE(X)/VALUE(Y);),
     TYPE(X) EQ. BOOLS (VX = VALUE(X);  VY = VALUE(Y);
                        VALUE(X) = IF +VX LE. +VY
                                   THEN VX OR.NOT.VY
                                   ELSE NOT.(NOT.VX AND.VY);),
     TYPE(X) EQ. SETS  (SYMDIFF(X, Y);),
     ERRTYPE(#DIVIDE(X,Y), X IS#, X););
     END FLOW;

RETURN;
END DIVIDE;


/*                              SYMDIFF                              */


DEFINE SYMDIFF(X, Y);

/* THIS SUBROUTINE SETS X = X/Y FOR X AND Y SETS (SYMMETRIC
DIFFERENCE).  IT IS CODED FROM THE RELATION X/Y = (X+Y)-(X*Y),
WHICH PROBABLY RESULTS IN A FASTER ROUTINE THAN (X-Y) + (Y-X).
THE LATTER WOULD REQUIRE COPYING Y, AS SETDIFF DESTROYS ITS
FIRST ARGUMENT.
     THIS ROUTINE RUNS FASTER IF Y IS THE SMALLER SET, BECAUSE
OF THE UNION X+Y. */

INTSECT(X, Y, R);          /* SET R = X*Y. */
AUNION(X, Y);              /* SET X = X + Y. */
SETDIFF(X, R);             /* SET X = X - R. */
RETURN;
END SYMDIFF;
```

6.22 DOUBLE SLASH ROUTINE

```
DEFINE DSLASH(X, Y);      /* ENTRY POINT FOR THE SETL X//Y. */

/* THIS SUBROUTINE SETS X = X//Y FOR THE TWO SETL MEANINGS OF
THE DOUBLE SLASH.  IF X AND Y ARE INTEGERS OR REALS, THE
RESULT IS THE REMAINDER AS DEFINED BY  X - TRUNC(X/Y)*Y  (AND
NOT  X - FLOOR(X/Y)*Y). */

IF TYPE(X) NE. TYPE(Y) THEN
  ERRMIX('DSLASH(X,Y), X, Y ='', X, Y);;

IF TYPE(X) EQ. INT  OR.  TYPE(X) EQ. REAL THEN
  /* REMAINDER. */
  VALUE(X) = VALUE(X)//VALUE(Y);
  RETURN;
  END IF;

IF TYPE(X) EQ. BOOL THEN
  /* EXCLUSIVE OR. */
  BOOLEX(X, Y);
  RETURN;
  END IF;

ERRTYPE('DSLASH(X,Y), X =''', X);
END DSLASH;
```

6.23 ABSOLUTE VALUE ROUTINE

```
DEFINE ABS(X);            /* ENTRY POINT FOR THE SETL ABS.X. */

/* THIS ROUTINE SETS X = ABS.X. */

IF TYPE(X) EQ. INT  OR.  TYPE(X) EQ. REAL THEN
  VALUE(X) = ABS.VALUE(X);
ELSE
  ERRTYPE('ABS.X, X =''', X);;

RETURN;
END ABS;
```

6.24 PARALLEL BOOLEAN OPERATIONS

```
DEFINE BOOLAND(A, B); /* ENTRY POINT FOR THE SETL A AND. B, */

IF TYPE(A) NE. BOOL THEN
  ERRTYPE(≠A AND. B, A IS NOT A BOOLEAN STRING:≠, A);;
IF TYPE(B) NE. BOOL THEN
  ERRTYPE(≠A AND. B, B IS NOT A BOOLEAN STRING:≠, B);;

VA = VALUE(A);
VB = VALUE(B);
IF ↓VA GE. ↓VB THEN
  VALUE(A) = (↓VA-↓VB)*≠0≠B + (VA(↓VA-↓VB+1:) AND. VB);
ELSE
  VALUE(A) = (↓VB-↓VA)*≠0≠B + (VA AND. VB(↓VB-↓VA+1↑));;

RETURN;
END BOOLAND;



/*                          BCOLOR                                  */

DEFINE BOOLOR(A, B);  /* ENTRY POINT FOR THE SETL A OR. B. */

IF TYPE(A) NE. BOOL THEN
  ERRTYPE(≠A OR. B, A IS NOT A BOOLEAN STRING↑≠, A);;
IF TYPE(B) NE. BOOL THEN
  ERRTYPE(≠A OR. B, B IS NOT A BOOLEAN STRING↑≠, B);;

VA = VALUE(A);
VB = VALUE(B);
IF ↓VA GE. ↓VB THEN
  VALUE(A) = VA(1↑↓VA-↓VB) + (VA(↓VA-↓VB+1:) OR. VB);
ELSE
  VALUE(A) = VB(1↑↓VB-↓VA) + (VA OR. VB(↓VB-↓VA+1↑));;

RETURN;
END BOOLOR;
```

```
/*                          BOOLEX                                  */

DEFINE BOOLEX(A, B); /* ENTRY POINT FOR THE SETL A EXOR. B, */

IF TYPE(A) NE. BOOL THEN
  ERRTYPE(≠A EXOR. B, A IS NOT A BOOLEAN STRING:≠, A);;
IF TYPE(B) NE. BOOL THEN
  ERRTYPE(≠A EXOR. B, B IS NOT A BOOLEAN STRING:≠, B);;
VA = VALUE(A);
VB = VALUE(B);
IF ↓VA GE. ↓VB THEN
  VALUE(A) = VA(1:↓VA-↓VB) + (VA(↓VA-↓VB+1:) EXOR. VB);
ELSE
  VALUE(A) = VB(1:↓VB-↓VA) + (VA EXOR. VB(↓VB-↓VA+1:));;
RETURN;
END BOOLEX;


/*                          BOOLIMP                                 */

DEFINE BOOLIMP(A, B); /* ENTRY POINT FOR SETL A IMPLIES. B, */

IF TYPE(A) NE. BOOL THEN
  ERRTYPE(≠A IMPLIES. B, A IS NOT A BOOLEAN STRING:≠, A);;
IF TYPE(B) NE. BOOL THEN
  ERRTYPE(≠A IMPLIES. B, B IS NOT A BOOLEAN STRING:≠, B);;
VA = VALUE(A);
VB = VALUE(B);
IF ↓VA GE. ↓VB THEN
  VALUE(A) = NOT.VA(1:↓VA-↓VB) + (VA(↓VA-↓VB+1:) IMPLIES. VB);
ELSE
  VALUE(A) = (↓VB-↓VA)*≠1≠B + (VA IMPLIES. VB(↓VB-↓VA+1:));;
RETURN;
END BOOLIMP;


/*                          BOOLNOT                                 */

DEFINE BOOLNOT(A);        /* ENTRY POINT FOR THE SETL NOT. A.*/

IF TYPE(A) NE. BOOL THEN
  ERRTYPE(≠NOT. A, A IS NOT A BOOLEAN STRING:≠, A);;
VALUE(A) = NOT. VALUE(A);
RETURN;
END BOOLNOT;
```

6.25 TYPE AND PAIR ROUTINES

```
DEFINEF SETLTYPE(X);      /* ENTRY POINT FOR THE SETL TYPE.X.*/

/* X IS THE OBJECT TO BE TESTED.  THE VALUE OF THIS FUNCTION
IS A SETL INTEGER GIVING THE TYPE CODE OF X, OR UNDEFWD IF X
IS UNDEFINED. */

IF TYPE(X) GT. SET THEN
  ERRTYPE(#TYPE.X, X IS#, X);;

IF X EQ. UNDEFWD THEN RETURN UNDEFWD;
ELSE RETURN <INT, TYPE(X)>;;

END SETLTYPE;



/*                          PAIR                                  */

DEFINEF PAIR(X);     /* ENTRY POINT FOR THE SETL PAIR. X. */

/* THE VALUE OF THIS FUNCTION IS TRUE IF X IS A PAIR AND FALSE
OTHERWISE.  A PAIR IS CONSIDERED TO BE A TUPLE WHOSE HIGHEST
INDEX IS TWO, EVEN IF ITS FIRST COMPONENT IS UNDEFINED. */

IF TYPE(X) EQ. TUPLE THEN
  IF NCOMPS(X) EQ. 2 THEN RETURN TRUE;;
RETURN FALSE;
END PAIR;
```

6.26 NEW ATOM ROUTINE

```
DEFINE NEWAT(R);

/* THIS SUBRCUTINE GENERATES A FRESH BLANK ATOM AND RETURNS
IT IN R.  THERE ARE NO INPUT ARGUMENTS.  THE VARIABLE
#BLANKATOM# IS GLOBAL AND IS INITIALIZED IN SECTION 5.1.8. */

VALUE(BLANKATOM) = VALUE(BLANKATOM) + 1;

IF VALUE(BLANKATOM) NE. 0 THEN R = BLANKATOM;
ELSE
  /* WRAP-ARCUND OCCURRED. */
  ERRMSG(#LIMIT EXCEEDED FOR BLANK ATOMS COUNTER.#);
  END IF;

RETURN;
END NEWAT;
```

6.27 MIN AND MAX ROUTINES

```
DEFINE MIN(X, Y, R);  /* ENTRY POINT FOR THE SETL X MIN. Y, */

IF TYPE(X) NE. TYPE(Y) THEN
  ERRMIX(≠X MIN. Y, X, Y =≠, X, Y);;

IF TYPE(X) EQ. INT.  OR.  TYPE(X) EQ. REAL THEN
  IF VALUE(X) LE. VALUE(Y) THEN R = X;
                           ELSE R = Y;;
  RETURN;
ELSE ERRTYPE(≠X MIN. Y, X =≠, X);;
END MIN;



/*                          MAX                                    */

DEFINE MAX(X, Y, R);  /* ENTRY POINT FOR THE SETL X MAX. Y, */

IF TYPE(X) NE. TYPE(Y) THEN
  ERRMIX(≠X MAX. Y, X, Y =≠, X, Y);;

IF TYPE(X) EQ. INT.  OR.  TYPE(X) EQ. REAL THEN
  IF VALUE(X) GE. VALUE(Y) THEN R = X;
                           ELSE R = Y;;
  RETURN;
ELSE ERRTYPE(≠X MAX. Y, X =≠, X);;
END MAX;
```

6.28 BOT AND TOP ROUTINES

```
DEFINEF BOT(X);      /* ENTRY POINT FOR THE SETL BOT. X, */

/* THE SETL BOT AND TOP OPERATORS CONVERT BACK AND FORTH
BETWEEN REALS AND INTEGERS AS ILLUSTRATED BELOW, ON A MACHINE
WHOSE FLOATING POINT IS DECIMAL WITH THREE SIGNIFICANT DIGITS.

          X              BOT. X          TOP. X

        3.00              3               3
        3.14              3               4
       -3.14             -4              -3
        3                 3.00            3.00
        3141              3.14E3          3.15E3
       -3141             -3.15E3         -3.14E3

ON A BINARY MACHINE, THE RESULTS WOULD BE SIMILAR BUT NOT
EXACTLY THE SAME IN CASES SUCH AS THE LAST TWO ROWS.  REGARD-
LESS OF MACHINE DETAILS, THE FOLLOWING RELATIONS HOLD FOR REAL
OR INTEGER X;

                    BOT. X ≤ X
                    TOP. X ≥ X
                    TOP. X = -BOT.-X          */

IF TYPE(X) EQ. REAL THEN
  R = VALUE(X)//1.0;      /* GET FRACTIONAL PART OF X, */
  IF R LT. 0 THEN R = R + 1.0;;
  /* ASSUME THAT  FIX  IS A FUNCTION THAT CONVERTS A FLOATING
  POINT INTEGER TO AN INTEGER. */
  RETURN <INT, FIX(VALUE(X)-R)>;
  END IF;

IF TYPE(X) EQ. INT THEN

  /* THE STEPS BELOW ASSUME THE EXISTENCE OF A  FLOAT
  FUNCTION WHICH PRODUCES EXACT RESULTS IF THE MAGNITUDE OF
  THE GIVEN INTEGER IS LESS THAN  BASE EXP. PRECISION  FOR
  THE MACHINE. */

  BASE = 2;
  PRECISION = 48;    /* USE (16,6) OR (16,14) FOR THE S/360, */
  MAX = BASE EXP. PRECISION;  /* AN INTEGER = MAXIMUM POSSIBLE
                                 FLOATING POINT FRACTION + 1. */
  VX = VALUE(X);
  F = 1.0;
  IF VX GE. 0 THEN
```

```
    (WHILE VX GE, MAX) VX = VX/BASE;  F = BASE*F;  END WHILE;
  ELSE
    D = BASE - 1;
    (WHILE VX LE, -MAX) VX = (VX-D)/BASE; F = BASE*F; END;
    END IF VX;
  RETURN <REAL, F*FLOAT(VX)>;
  END IF TYPE;

ERRTYPE(#BOT, X, X IS NEITHER A REAL NOR AN INTEGER!#, X);
END BOT;



/*                          TOP                                    */


DEFINEF TOP(X);       /* ENTRY POINT FOR THE SETL TOP, X, */

PMINUS(X);            /* TOP(X) = -BOT(-X). */
R = BOT(X);
PMINUS(R);
PMINUS(X);            /* (RESTORE THE ARGUMENT X). */
RETURN R;
END TOP;
```

6.29 SUBSTRING ROUTINES

THE ROUTINES IN THIS SECTION IMPLEMENT THE SETL EXPRESSION S(I:L) FOR BOTH RETRIEVAL AND STORAGE. WE ASSUME THAT I AND L ARE ALWAYS DEFINED, THAT IS, THE COMPILER TRANSLATES S(I:) TO S(I:↓S-I+1) AND S(:L) TO S(1:L), IN BOTH RETRIEVAL AND STORAGE CONTEXTS. THE MEANING OF THE STORAGE CONTEXT, S(I:L) = R, IS TAKEN TO BE

S = S(1:I-1) + R + S(I+L:↓S-(I+L-1)).

INTUITIVELY, THE SUBSTRING S(I:L) IS REPLACED BY R, AND THE PORTION OF S TO THE RIGHT OF S(I:L) IS SLID LEFT OR RIGHT, AS REQUIRED, TO MAKE ROOM FOR R.

NOTE THAT THE SUBSTRING OPERATION IS NOT A PURE RETRIEVAL OPERATION (THAT IS, AFTER S(I:L) = R, THE VALUE OF S(I:L) IS NOT NECESSARILY R), EXCEPT IN CERTAIN SPECIAL CASES, MOST NOTABLY WHEN L = ↓R.

```
DEFINEF SUBSTR(S, I, L); /* ENTRY POINT FOR THE SETL S(I:L).*/

IF TYPE(S) N,→≤BOOL, CHAR, TUPLE≥ THEN
  ERRTYPE(≠S(I:L), S, NOT A STRING OR TUPLE, IS:≠, S);;

IF TYPE(I) NE, INT THEN
  ERRTYPE(≠S(I:L), I, NOT AN INTEGER, IS:≠, I);;

IF TYPE(L) NE, INT THEN
  ERRTYPE(≠S(I:L), L, NOT AN INTEGER, IS:≠, L);;

VS = VALUE(S);
VI = VALUE(I);
VL = VALUE(L);
IF VI LE, 0 THEN ERRVAL(≠S(I:L), I ≤ 0:≠, I);;

IF VL LT, 0 THEN ERRVAL(≠S(I:L), L < 0:≠, L);;

ILAST = VI + VL - 1;
IF ILAST GT, (IF TYPE(S) EQ, TUPLE THEN #COMPS(S) ELSE ↓VS)
  THEN ERRVAL(≠S(I:L), I + L - 1 > ↓S:≠, ILAST);;
```

```
/* NOW IT IS KNOWN THAT:

                        I ≥ 1
                        L ≥ 0
                        I + L - 1 ≤ ↓S.

FROM THIS IT FOLLOWS THAT I ≤ ↓S EXCEPT POSSIBLY IF L = 0, IN
WHICH CASE I MAY BE ↓S+1.  THIS IS CONSIDERED VALID. TO ALLOW
CODE SEQUENCES SUCH AS
          (0 ≤ ↑L ≤ ↓S) ... S(↓S-L+1:L) ...
TO REFERENCE THE RIGHT SUBSTRINGS OF S, INCLUDING THE NULL
STRING.  SIMILARLY, IT FOLLOWS FROM THE ABOVE THAT L ≤ ↓S, SO
THIS NEED NOT BE EXPLICITLY CHECKED FOR. */

IF TYPE(S) EQ. BOOL THEN
  IF VL EQ. 0 THEN RETURN NULLBSTR;
              ELSE RETURN <BOOL, (VALUE(S))(VI:VL)>;;;

IF TYPE(S) EQ. CHAR THEN
  IF VL EQ. 0 THEN RETURN NULLCSTR;
              ELSE RETURN <CHAR, (VALUE(S))(VI:VL)>;;;

/* TYPE(S) = TUPLE. */

IF VL EQ. 0 THEN RETURN NULLTUPLE;;

/* IGNOR TRAILING UNDEFINED COMPONENTS IN SUBTUPLE. */
NEWL = VL;
(WHILE (TUP(S))(NEWL) EQ. UNDEFWD)
  NEWL = NEWL - 1;
  IF NEWL EQ. 0 THEN RETURN NULLTUPLE;;
  END WHILE;

RETURN <TUPLE, <NEWL, NCOMPALLC(NEWL), (VALUE(S))(VI:VL)>>;
END SUBSTR;
```

```
/*                          SSUBSTR                                  */

DEFINE SSUBSTR(S, I, L, R);   /* ENTRY POINT FOR S(I:L) = R.*/

IF TYPE(S) N,→≤BOOL, CHAR, TUPLE≥ THEN
  ERRTYPE(≠S(I:L) = R, S, NOT A STRING OR TUPLE, IS:≠, S);;

IF TYPE(I) NE, INT THEN
  ERRTYPE(≠S(I:L) = R, I, NOT AN INTEGER, IS:≠, I);;

IF TYPE(L) NE, INT THEN
  ERRTYPE(≠S(I:L) = R, L, NOT AN INTEGER, IS:≠, L);;

VS = VALUE(S);
VI = VALUE(I);
VL = VALUE(L);
VR = VALUE(R);

IF VI LE, 0 THEN ERRVAL(≠S(I:L) = R, I ≤ 0:≠, I);;

IF VL LT, 0 THEN ERRVAL(≠S(I:L) = R, L ≤ 0:≠, L);;

ILAST = VI + VL - 1;
IF ILAST GT, (IF TYPE(S) EQ, TUPLE THEN NCOMPS(S) ELSE ↓VS)
  THEN ERRVAL(≠S(I:L) = R, I + L - 1 > ↓S:≠, ILAST);;

/* THE CONDITIONS FOR THE VALIDITY OF I AND L HAVE BEEN MADE
THE SAME AS IN THE CASE OF THE RETRIEVAL SUBSTRING OPERATION,
FOR LITTLE REASON OTHER THAN CONSISTENCY.  IT WOULD PROBABLY
BE REASONABLE TO ALLOW L TO BE ARBITRARILY LARGE FOR THE
STORAGE SUBSTRING OPERATION, BUT THAT IS NOT DONE. */

IF TYPE(S) NE, TYPE(R) THEN
  ERRMIX(≠S(I) = R (SSUBSTR), S, R =≠, S, R);;

/* THERE IS NO RESTRICTION ON THE LENGTH OF R. */

S1 = <TYPE(S), SUBSTR(S, <INT,1>, <INT,VI-1>)>;
S2 = <TYPE(S), SUBSTR(S, <INT,VI+VL>, <INT, (IF TYPE(S) EQ,
                TUPLE THEN NCOMPS(S) ELSE ↓VS)-ILAST>)>;
PLUS(S1, R);          /* S1 = S(1:I-1) + R. */
PLUS(S1, S2);         /* S1 = S(1:I-1) + R + S(I+L:↓S-(I+L-1))*/
S = S1;
RETURN;
END SSUBSTR;
```

6.30 DEC AND OCT ROUTINES

```
DEFINEF DEC(S);      /* ENTRY POINT FOR THE SETL DEC. X, */

/* THIS FUNCTION CONVERTS A CHARACTER STRING, CONTAINING THE
DECIMAL REPRESENTATION OF A NUMBER, TO AN INTEGER, AND VICE
VERSA. */

IF TYPE(X) EQ. INT THEN RETURN CHARINT(X, 0, 10);;
IF TYPE(X) EQ. CHAR THEN RETURN INTCHAR(X, 10);;
ERRTYPE(#DEC, X, X IS NEITHER AN INTEGER NOR A CHARACTER #
      + #STRING;#, X);

/* THE LOWER LEVEL ROUTINES CHARINT AND INTCHAR ARE NOT SHOWN
IN THIS SPECIFICATION.  CHARINT(X,W,B) CONVERTS THE INTEGER
X TO A CHARACTER STRING OF LENGTH W, TO BASE B.  IF X < 0, A
MINUS SIGN IS PLACED AT THE LEFT.  IF W = 0, THE RESULTING
STRING IS JUST LONG ENOUGH TO HOLD THE RESULT, E.G., #-23#,
#99#, ETC.  IF W IS NONZERO, THE RESULT IS PADDED WITH BLANKS
OR TRUNCATED ON THE LEFT, AS REQUIRED.
     INTCHAR(X,B) CONVERTS THE CHARACTER STRING X TO AN
INTEGER, INTERPRETING THE STRING AS BASE B. */

END DEC;



/*                          OCT                                  */

DEFINEF OCT(X);      /* ENTRY POINT FOR THE SETL OCT. X, */

/* THIS FUNCTION CONVERTS A CHARACTER STRING, CONTAINING THE
OCTAL REPRESENTATION OF A NUMBER, TO AN INTEGER, AND VICE
VERSA. */

IF TYPE(X) EQ. INT THEN RETURN CHARINT(X, 0, 8);;
IF TYPE(X) EQ. CHAR THEN RETURN INTCHAR(X, 8);;
ERRTYPE(#OCT, X, X IS NEITHER AN INTEGER NOR A CHARACTER #
      + #STRING;#, X);

/* SEE NOTE IN ROUTINE #DEC#, ABOVE. */

END OCT;
```

6.31 BITR ROUTINE

```
DEFINEF BITR(X);     /* ENTRY POINT FOR THE SETL BITR, X. */

/* THIS ROUTINE CONVERTS X AS FOLLOWS:
     INTEGER TO BOOLEAN STRING
     REAL TO BOOLEAN STRING (BITR(BOT(X))
     BOOLEAN STRING TO INTEGER.  */

IF TYPE(X) ∈ {INT, REAL} THEN
  VX = IF TYPE(X) EQ. INT THEN VALUE(X) ELSE VALUE(BOT(X));
  IF VX GE. 0 THEN
    R = NULT;
    (WHILE VX NE. 0) R = (IF VX//2 EQ. 1 THEN 1B ELSE 0B) + R;
      VX = VX / 2;
      END WHILE;
    RETURN <BOOL, R>;     /* NOTE THAT BITR(0) = NULLBSTR. */
  ELSE ERRVAL('BITR. X, X < 0:', X);
    END IF VX GE. 0;
  END IF TYPE(X);

IF TYPE(X) EQ. BOOL THEN
  VX = VALUE(X);
  R = 0;
  (1 ≤ ∨I ≤ #VX)  R = 2*R + VX(I);;
  RETURN <INT, R>;
  END IF;

ERRTYPE('BITR. X, X IS NEITHER AN INTEGER, A REAL, NOR A '
      + 'BOOLEAN STRING:', X);
END BITR;
```

6.32 RELATIONAL OPERATORS LE AND LT

```
DEFINEF LE(X,Y);     /* ENTRY POINT FOR THE SETL X LE. Y. */

/* THE RESULT IS DEFINED FOR INTEGERS, REALS, BOOLEAN STRINGS.
AND SETS.  THE RESULT IS ALWAYS EITHER TRUE OR FALSE.  THERE
ARE NO ROUTINES FOR GE AND GT, AS THE COMPILER MAY MAKE THE
TRANSLATION GE(X,Y) = LE(Y,X) AND GT(X,Y) = LT(Y,X), WHICH
IS VALID FOR ALL DATA TYPES. */

IF TYPE(X) NE. TYPE(Y) THEN
  ERRMIX(#X LE. Y OR X GE. Y, X, Y =#, X, Y);;

IF TYPE(X) EQ. INT  THEN RETURN <BOOL,VALUE(X) LE. VALUE(Y)>;;
IF TYPE(X) EQ. REAL THEN RETURN <BOOL,VALUE(X) LE. VALUE(Y)>;;
IF TYPE(X) EQ. BOOL THEN
  /* THE MEANING IS #X IMPLIES Y EVERYWHERE.# */
  VX = VALUE(X);
  VY = VALUE(Y);
  I = #VX;
  J = #VY;
  (WHILE I GE. 1 AND. J GE. 1)
    IF VX(I) EQ. 1B AND. VY(J) EQ. 0B THEN RETURN FALSE;;
    I = I - 1;
    J = J - 1;
    END WHILE;

  IF I EQ. 0 THEN RETURN TRUE;;     /* (#VX ≤ #VY). */
  /* X IS THE LONGER STRING.  LOOK FOR 1#S IN ITS LEFT PART.*/
  (WHILE I GE. 1)
    IF VX(I) EQ. 1B THEN RETURN FALSE;;
    I = I - 1;
    END WHILE;
  /* LEFT PART OF X WAS ALL ZEROS. */
  RETURN TRUE;
  END IF TYPE(X) EQ. BOOL;

IF TYPE(X) EQ. SET THEN
  /* THE MEANING IS #X IS A SUBSET OF Y#. */
  IF NMEMBS(X) GT. NMEMBS(Y) THEN RETURN FALSE;;
  C = 0;
LOOP:
    M = NEXTMEM(C, X);
    IF M EQ. UNDEFWD THEN RETURN TRUE;;
    IF ELMTSET(M, X) EQ. FALSE THEN RETURN FALSE;;
    GO TO LOOP;
  END IF TYPE(X) EQ. SET;
ERRTYPE(#X LE. Y OR X GE. Y, X =#, X);
END LE;
```

```
/*                              LT                                    */

DEFINEF LT(X, Y);    /* ENTRY POINT FOR THE SETL X LT. Y. */

/* THE RELATION X LT. Y = NOT.(Y LE. X) HOLDS FOR INTEGERS AND
REALS ONLY.  IT DOES NOT HOLD FOR BOOLEAN STRINGS AND SETS,
THE RELATION X LT. Y = (X LE. Y AND. X NE. Y) HOLDS FOR
INTEGERS, REALS, AND SETS, BUT NOT FOR BOOLEAN STRINGS.  FOR
SETS WE USE THE RELATION X LT. Y = (↓X < ↓Y AND. X LE. Y. */

IF TYPE(X) NE. TYPE(Y) THEN
  ERRMIX(≠X LT. Y OR X GT. Y, X, Y =≠, X, Y);;

IF TYPE(X) → ≤INT, REAL≥ THEN
  IF LE(Y,X) EQ. TRUE THEN RETURN FALSE;
  ELSE RETURN TRUE;;

IF TYPE(X) EQ. BOOL THEN
  /* THE MEANING IS ≠X IS 0B AND Y IS 1B EVERYWHERE≠. */
  VY = VALUE(Y);
  IF ≡B → VY ↑ B EQ. 0B THEN RETURN FALSE;;
  /* NOW Y IS KNOWN TO BE ALL 1≠S (OR NULL). */
  VX = VALUE(X);
  IF ↓VX GT. ↓VY THEN RETURN FALSE;;    /* (Y WOULD BE LEFT-
                    EXTENDED WITH ZEROS). */
  IF ≡B → VX ↑ B EQ. 1B THEN RETURN FALSE;;
  /* NOW X IS KNOWN TO BE ALL ZEROS, AND ↓VX ≤ ↓VY. */
  RETURN TRUE;       /* X AND Y COULD BE NULL. */
  END IF TYPE(X) EQ. BOOL;

IF TYPE(X) EQ. SET THEN
  /* THE MEANING IS ≠X IS A PROPER SUBSET OF Y≠. */
  IF NMEMBS(X) LT. NMEMBS(Y) THEN RETURN LE(X,Y);
  ELSE RETURN FALSE;;
  END IF TYPE(X) EQ. SET;
ERRTYPE(≠X LT. Y OR X GT. Y, X =≠, X);
END LT;
```

6.33 POWER SET ROUTINES

THE ROUTINES IN THIS SECTION IMPLEMENT THE SETL POWER SET POW(S) AND THE POWER SET RESTRICTED TO SUBSETS OF SIZE N, NPOW(N,S). THE ROUTINES ARE CODED MORE FOR CONCISENESS THAN FOR SPEED, BECAUSE THEY ARE OF FAIRLY LOW USE AND BECAUSE THEY CANNOT BE USED FOR VERY LARGE SETS (MORE THAN TEN MEMBERS OR SO FOR POW), AS THE USER WOULD RAPIDLY RUN OUT OF HEAP SPACE.

THE MOST CONCISE ALGORITHM FOR NPOW(N,S) KNOWN TO THE AUTHOR IS THE FOLLOWING RECURSIVE ONE THAT BEARS A RELATION TO PASCAL'S TRIANGLE (IN WHICH P(N,M) = P(N,M-1) + P(N-1,M-1));

```
DEFINEF NPOW(N, S);
IF N GT. #S  OR.  N LT. 0 THEN RETURN NL.;
IF S EQ. NL. THEN RETURN {NL.};
X = ARB. S;
R = S LESS. X;
RETURN NPOW(N,R) + {M WITH. X, M + NPOW(N-1,R)};
END NPOW;
```

HOWEVER, THIS ALGORITHM IS NOT USED, BECAUSE WE FAVOR ONE THAT RESTS ON AN UNDERLYING ROUTINE THAT RETURNS SUCCESSIVE MEMBERS OF THE POWER SET ON EACH CALL. THIS PERMITS ITERATION OVER A POWER SET WITHOUT ACTUALLY CONSTRUCTING THE WHOLE POWER SET. THIS FACILITY WOULD ONLY BE AVAILABLE TO THE SETL USER BY GOING OUT OF SETL. ALTERNATIVELY, THE SETL COMPILER COULD INCORPORATE THE SPECIAL CASE OPTIMIZATION OF LOOKING FOR THE ITERATIONS (∀X∈POW(S)) AND (∀X∈NPOW(N,S)), AND TRANSLATING THEM TO CALLS ON THE LOWER LEVEL ROUTINES.

THERE ARE FOUR ROUTINES IN ALL, AS FOLLOWS:

POW(S)	COMPUTES THE WHOLE POWER SET.
NPOW(N,S)	COMPUTES THE SET OF ALL N-MEMBER SUBSETS.
NEXPOW(C,S)	GETS NEXT MEMBER OF POW(S).
NEXNPOW(C,N,S)	GETS NEXT MEMBER OF NPOW(N,S).

THE FUNDAMENTAL ONE IS NEXNPOW(C,N,S), AND THE OTHERS ARE TRIVIALLY BUILT ON THIS.

WE ALLOW N TO BE "OUT OF RANGE" IN NPOW(N,S). THIS IS BECAUSE IT MIGHT BE USEFUL IN CERTAIN CASES SUCH AS IN THE ALGORITHM ABOVE. SOME SPECIAL CASES:

```
NPOW(N,S) = NL. (NOT {NL.}) IF N < 0 OR N > #S.
NPOW(0,S) = {NL.},
NPOW(1,S) = {{M}, M + S},
NPOW(#S,S) = {S},
POW(NL.) = {NL.},
```

```
     POW(≤A≥) = ≤NL., ≤A≥≥,    */


     /*                          POW                                  */

     DEFINEF POW(S);      /* ENTRY POINT FOR THE SETL POW(S), */

     /* THIS FUNCTION CALCULATES THE POWER SET OF S (SET OF ALL
     SUBSETS). */

     IF TYPE(S) NE. SET THEN ERRTYPE(≠POW(S), S IS NOT A SET:≠,S);;

     R = NULLSET;
     C = 0;
LOOP:  M = NEXPOW(C,S);
       IF M EQ. UNDEFWD THEN RETURN R;;
       AUGAOK(M,R);
       GO TO LOOP;
     END POW;


     /*                          NPOW                                 */

     DEFINEF NPOW(N, S); /* ENTRY POINT FOR THE SETL NPOW(N,S). */

     /* THIS FUNCTION CALCULATES THE SET OF ALL SUBSETS OF S OF
     SIZE N. */

     IF TYPE(N) NE. INT THEN ERRTYPE(≠NPOW(N,S), N IS NOT AN ≠
                                  + ≠INTEGER:≠, N);;
     IF TYPE(S) NE. SET THEN ERRTYPE(≠NPOW(N,S), S IS NOT A SET:≠,
                                                          S);;

     R = NULLSET;
     C = 0;
LOOP:  M = NEXNPOW(C, N, S);
       IF M EQ. UNDEFWD THEN RETURN R;;
       AUGAOK(M, R);
       GO TO LOOP;
     END NPOW;
```

```
/*                          NEXPOW                              */

DEFINEF NEXPOW(C, S);

/* GIVEN A CONTROL ITEM C AND A SET S, THIS FUNCTION RETURNS
WITH THE ≠NEXT≠ SUBSET OF S.  THE CALLER SHOULD HAVE C = 0
ON FIRST ENTRY, AND SHOULD NOT ALTER C THEREAFTER.  COMPLETION
OF THE PROCESS IS INDICATED BY THE ROUTINE≠S RETURNING AN
UNDEFINED VALUE.
     THE ROUTINE FIRST RETURNS THE NULL SET, THEN SUBSETS OF
SIZE ONE, THEN THOSE OF SIZE TWO, ..., THEN A COPY OF S
ITSELF, AND FINALLY THE UNDEFINED VALUE.  (THE FACT THAT IT
WORKS IN THIS ORDER IS COINCIDENTAL AND NOT PART OF THE SETL
LANGUAGE DESIGN).  */

IF C EQ. 0 THEN
  /* FIRST ENTRY.  INITIALIZE C. */
  C = <TUPLE, <2,2,<0,0>>>;    /* C IS A PAIR (C1,N), WHERE */
  END IF;                      /*  C1 AND N ARE ARGUMENTS TO */
                               /*  NEXTNPOW(C1,N,S).  */

/* USUAL CASE. */
C1 = (TUP(C))(1);
N = (TUP(C))(2);
M = NEXNPOW(C1,N,S);
IF M EQ. UNDEFWD THEN
  /* THROUGH WITH SUBSETS OF SIZE N. */
  IF N EQ. NMEMBS(S) THEN RETURN UNDEFWD;  /* ALL DONE. */
  N = N + 1;           /* SET FOR LARGER SUBSETS. */
  (TUP(C))(2) = N;     /* SAVE NEW N FOR NEXT ENTRY. */
  C1 = 0;
  M = NEXNPOW(C1, N, S);
  END IF;

(TUP(C))(1) = C1;    /* SAVE C1 FOR NEXT ENTRY. */
RETURN M;
END NEXPOW;
```

```
DEFINEF NEXNPOW(C, N, S);

/* THIS ROUTINE FINDS THE *NEXT* SUBSET OF S OF SIZE N, AFTER
THE ONE INDICATED BY THE CONTROL ITEM C.  ON FIRST ENTRY, THE
CALLER SHOULD HAVE C = THE INTEGER ZERO.  ON SUBSEQUENT
ENTRIES, THE CALLER SHOULD NOT ALTER C.  COMPLETION OF THE
PROCESS IS INDICATED BY THIS ROUTINE*S RETURNING AN UNDEFINED
VALUE.
     THE ALGORITHM WORKS BY FIRST ARRANGING THE MEMBERS OF S
INTO A VECTOR U (IN AN ARBITRARY ORDER) OF LENGTH #S.  AN
AUXILARY VECTOR V OF INTEGERS <1,2,...,N> IS ALSO MADE UP.  ON
EACH ENTRY, THE LAST POSSIBLE COMPONENT OF V IS INCREASED BY
THE LEAST POSSIBLE AMOUNT.  FOR EXAMPLE, IF N = 3 AND #S = 5,
THE SUCCESSIVE VALUES OF V ARE:

          <1,2,3>        <1,4,5>
          <1,2,4>        <2,3,4>
          <1,2,5>        <2,3,5>
          <1,3,4>        <2,4,5>
          <1,3,5>        <3,4,5>

THE CONTROL ITEM C IS A PAIR (U,V), WHERE U IS THE VECTOR OF
MEMBERS OF S, AND V IS THE VECTOR OF INTEGERS. */

VN = VALUE(N);
SIZES = NMEMBS(S);
IF VN LT. 0  OR.  VN GT. SIZES THEN RETURN UNDEFWD;;
IF C EQ. 0 THEN
  /* FIRST ENTRY.  BUILD U AND V. */
  U = <TUPLE, <SIZES, SIZES, NULT.>>;
  I = 0;
  C1 = 0;

LOOP:   M = NEXTMEM(C1, S);
        IF M EQ. UNDEFWD THEN GO TO OUT;;
        I = I + 1;
        (TUP(U))(I) = M;
        GO TO LOOP;

OUT;    V = <TUPLE, <N, N, NULT.>>;
        (1 ≤ ~I ≤ N) (TUP(V))(I) = I;;
        C = <TUPLE, <2, 2, <U,V>>>;
      ELSE
        /* SUBSEQUENT ENTRY.  EXTRACT U AND V, AND INCREMENT V. */
        U = (TUP(C))(1);
        V = (TUP(C))(2);
        /* IF POSSIBLE, INCREMENT THE LAST COMPONENT OF V. */
```

```
      IF N EQ. 0 THEN RETURN UNDEFWD;;
      IF (TUP(V))(N) LT. SIZES THEN
        (TUP(V))(N) = (TUP(V))(N) + 1;
      ELSE
        /* SEARCH V, RIGHT TO LEFT, FOR A COMPONENT THAT CAN BE
      INCREMENTED. */
        (N > ~I ≥ 1)
          IF (TUP(V))(I) LT. ((TUP(V))(I+1)-1) THEN
            /* INCREMENT I≠TH COMPONENT, AND RESET ALL TO RIGHT.*/
            (I ≤ ~J ≤ N) (TUP(V))(J) = (TUP(V))(I) + 1 + J - I;;
            GO TO MAKE;
            END IF;
          END ~I;
        /* NO COMPONENT OF V CAN BE INCREASED.  THROUGH. */
        RETURN UNDEFWD;
        END IF (TUP(V))(N);
      END IF C EQ. 0;

    /* MAKE THE SUBSET OF S CORRESPONDING TO V. */
MAKE; R = NULLSET;
    (1 ≤ ~I ≤ N) AUGAOK((TUP(U))((TUP(V))(I)), R);;
    RETURN R;
    END NEXNPOW;
```

6.34 RANDOM ROUTINES

THE ROUTINES IN THIS SECTION IMPLEMENT THE SETL RANDOM. X. THE ROUTINE NAMES AND MEANINGS FOR THE VARIOUS DATA TYPES FOR X ARE:

RANDOM(X)	MAIN ROUTINE
RANINT(N)	N A SETL INTEGER, RETURNS I, 1 ≤ I ≤ N, UNIFORMLY DISTRIBUTED
RANREAL(R)	R REAL, RETURNS X, 0 ≤ X < R, UNIFORM
RANBOOL(B)	B A BOOLEAN STRING, RETURNS B(I), 1 ≤ I ≤ #B, I UNIFORM
RANCHAR(C)	C A CHARACTER STRING, RETURNS C(I), 1 ≤ I ≤ #C, I UNIFORM
RANTUPL(T)	T A TUPLE, RETURNS T(I), 1 ≤ I ≤ #T, I UNIFORM
RANSET(S)	S A SET, RETURNS A MEMBER M∈S, M UNIFORMLY DISTRIBUTED OVER S
RANBASE(N)	N AN ORDINARY (NON-SETL) INTEGER, RETURNS I, 0 ≤ I < N, UNIFORMLY DISTRIBUTED.

```
DEFINEF RANDOM(X);  /* ENTRY POINT FOR THE SETL RANDOM, X. */

FLOW TYPE(X) EQ, INT$  (RETURN RANINT(X);),
     TYPE(X) EQ, REAL$ (RETURN RANREAL(X);),
     TYPE(X) EQ, BOOL$ (RETURN RANBOOL(X);),
     TYPE(X) EQ, CHAR$ (RETURN RANCHAR(X);),
     TYPE(X) EQ, TUPL$ (RETURN RANTUPL(X);),
     TYPE(X) EQ, SET$  (RETURN RANSET(X);),
     (ERRTYPE(#RANDOM, X, X IS:#, X););
     END FLOW;
END RANDOM;
```

```
/*                          RANINT                              */

DEFINEF RANINT(N);

/* N IS A SETL INTEGER, WHICH MUST BE ≥ 1.  THE RESULT IS A
SETL INTEGER UNIFORM ON [1,N]. */

VN = VALUE(N);
IF VN LE. 0 THEN ERRVAL(#RANDOM.N, N ≤ 0:#, N);;

/* MOST OF THE STEPS BELOW HAVE TO DO WITH THE FACT THAT N
CAN EXCEED THE RANGE OF THE BASIC RANDOM NUMBER GENERATOR,
RANBASE.  THE GLOBAL VARIABLE MAXRNP1 IS THE MAXIMUM RANDOM
NUMBER RETURNABLE FROM RANBASE, PLUS ONE (RECALL THAT
0 ≤ RANBASE(N) ≤ N-1). */

R = RANBASE(VN//MAXRNP1);      /* INITIALIZE THE RESULT R. */
K = VN/MAXRNP1;              /* NUMBER OF TIMES FOR LOOP BELOW. */
(1 ≤ ~I ≤ K)                 /* USUALLY K = 0. */
  R = R*MAXRNP1 + RANBASE(MAXRNP1);
  END ~I;
RETURN <INT, R+1>;
END RANINT;



/*                          RANREAL                             */

DEFINEF RANREAL(R);

/* R IS A REAL, AND MUST BE > 0.  THE RESULT IS A REAL UNIFORM
ON [0,R). */

VR = VALUE(R);
IF VR LE. 0 THEN ERRVAL(#RANDOM. R, R ≤ 0:#, R);;

/* WE ASSUME THE EXISTENCE OF A FLOAT FUNCTION THAT CONVERTS
AN INTEGER TO FLOATING POINT, TRUNCATING ON THE RIGHT IF
NECESSARY. */

RETURN <REAL, VR*FLOAT(RANBASE(MAXRNP1))/FLOAT(MAXRNP1)>;
END RANREAL;
```

```
/*                          RANBOOL                          */

DEFINEF RANBOOL(B);

/* B IS A BOOLEAN STRING,  THE RESULT IS A RANDOM BIT FROM B,
UNIFORMLY DISTRIBUTED OVER THE LENGTH OF B.  IF B IS NULL, THE
RESULT IS UNDEFINED. */

IF B EQ. NULLBSTR THEN RETURN UNDEFWD;
ELSE RETURN CFBSTR(B, RANINT(↓VALUE(B)));;
END RANBOOL;


/*                          RANCHAR                          */

DEFINEF RANCHAR(C);

/* C IS A CHARACTER STRING,  THE RESULT IS A RANDOM CHARACTER
FROM C, UNIFORMLY DISTRIBUTED OVER THE LENGTH OF C.  IF C IS
NULL, THE RESULT IS UNDEFINED. */

IF C EQ. NULLCSTR THEN RETURN UNDEFWD;
ELSE RETURN CFCSTR(C, RANINT(↓VALUE(C)));;
END RANCHAR;


/*                          RANTUPL                          */

DEFINEF RANTUPL(T);

/* T IS A TUPLE,  THE RESULT IS A RANDOM COMPONENT FROM T,
UNIFORMLY DISTRIBUTED OVER ITS LENGTH.  IF T IS SPARSE, THEN
UNDEFINED RESULTS WILL PREDOMINATE IN PROPORTION TO ITS
SPARSENESS. */

IF T EQ. NULLTUPLE THEN RETURN UNDEFWD;
ELSE RETURN CFTUPLE(T, RANINT(NCOMPS(T)));;
/* THIS IS NOT VERY EFFICIENT, BUT THIS WILL PROBABLY BE A
VERY LOW USE ROUTINE, SO IT'S BETTER TO SAVE SPACE IN THE RUN
TIME LIBRARY. */
END RANTUPL;
```

```
/*                          RANSET                              */

DEFINEF RANSET(S);

/* S IS A SET.  THE RESULT IS A RANDOM MEMBER OF S, UNIFORMLY
DISTRIBUTED OVER S.  IF S IS NULL, THE RESULT IS UNDEFINED.
     THIS ROUTINE IS DIFFICULT TO IMPLEMENT EFFICIENTLY.  ONE
WOULD LIKE TO BEGIN PROBING AT A UNIFORMLY RANDOM HASH TABLE
ENTRY.  BUT THIS WOULD NOT RESULT IN A UNIFORM DISTRIBUTION OF
RESULTS, BECAUSE FOR A GIVEN SET THE USED ENTRIES ARE NOT
UNIFORMLY DISTRIBUTED, EACH ENTRY MAY CONTAIN A LIST OF
ARBITRARY LENGTH, AND A MEMBER IN A LIST MAY BE A SPECIAL
PAIR, WHICH REPRESENTS MANY MEMBERS.
     FOR SOME ALGORITHMS IT MAY BE VERY IMPORTANT THAT THE
RESULTS ARE UNIFORMLY DISTRIBUTED.  THEREFORE, WITH THIS IN
MIND AND GUESSING THAT THIS WILL BE A FAIRLY LOW-USE ROUTINE,
WE TAKE THE VERY SIMPLE BUT INEFFICIENT APPROACH OF GENERATING
A RANDOM INTEGER N LESS THAN THE SIZE OF S, AND STEPPING
THROUGH S N TIMES, USING NEXTMEM. */

IF S EQ. NULLSET THEN RETURN UNDEFWD;;

N = RANBASE(NMEMBS(S));  /* 0 ≤ N < NMEMBS(S). */
C = 0;
(0 ≤ ~I ≤ N) M = NEXTMEM(C, S);;
RETURN M;
END RANSET;
```

```
DEFINEF RANBASE(K);

/* K IS AN ORDINARY (NON-SETL) INTEGER.  THE RESULT IS AN
ORDINARY INTEGER FROM 0 TO K-1, UNIFORMLY DISTRIBUTED.
     THIS IS THE TAUSWORTHE GENERATOR FOR A 32 BIT MACHINE
(SIGN AND 31 MAGNITUDE).  THE SEQUENCE IS OF MAXIMUM LENGTH
FOR THE WORD SIZE USED.
     REFERENCES:

     1.  TAUSWORTHE, ROBERT C., MATHEMATICS OF COMPUTATION
         1965 PAGES 201-209.
     2.  WHITTLESLEY, J., CACM SEPTEMBER 1968 PAGES 641-644.
     3.  PAYNE, W. H., CACM JANUARY 1970 PAGE 57.

     THE ROUTINE USES PARAMETERS N AND M, WHICH ARE CHOSEN
BASED ON THE MACHINE'S WORD SIZE.  ONE NORMALLY CHOOSES N
EQUAL TO THE WORD SIZE LESS ONE, AND THEN M AS FOLLOWS:

          N        M
          -------------
          11    2
          15    1, 4, OR 7
          17    3, 5, OR 6
          23    5 OR 9
          31    3, 6, 7, OR 13
          63    1, 5, OR 31

     THE CALCULATION OF SUITABLE VALUES OF M FOR A GIVEN N
INVOLVES FINDING PRIMITIVE POLYNOMIALS; SEE (1) PAGE 208.
     THIS ROUTINE USES A GLOBAL VARIABLE #RANDSEED# (SEE
SECTION 5.1.8), WHICH MUST BE INITIALIZED TO A BOOLEAN STRING,
NOT ALL ZEROS, OF LENGTH N.  */
```

```
        N = 31;                         /* CONSTANTS. */
        M = 13;

        /* FIRST UPDATE THE VALUE OF #RANDSEED#. */

AGAIN:  B = RANDSEED(1:(↓RANDSEED-M));      /* RIGHT SHIFT M. */
        A = RANDSEED EXOR. B;
        B = A(N-M+1:M) * (N-M)*0B;          /* LEFT SHIFT N-M. */
        RANDSEED = A EXOR. B;

        /* NOW CONVERT #RANDSEED# TO AN INTEGER RANGING FROM 0 TO
        K-1, WHERE K < 2 EXP. N.  THIS IS DONE BY TRUNCATING
        #RANDSEED# TO THE APPROPRIATE NUMBER OF BITS.  IF THE RESULT
        IS LESS THAN K, IT IS RETURNED.  OTHERWISE THE ROUTINE STARTS
        ALL OVER AGAIN.  THIS GUARANTEES A UNIFORMLY DISTRIBUTED
        RESULT.  THE PROCESS MUST TERMINATE, AS #RANDSEED# IS OF
        MAXIMUM PERIOD (2 EXP. N - 1). */

        MASK = (↓BITR.K)*1B;                /* ALL ONES, LENGTH OF K.*/
        RESULT = BITR.(RANDSEED AND. MASK);
        IF RESULT GT. K THEN GO TO AGAIN;
        ELSE RETURN RESULT - 1;;
        END RANBASE;
```

6.35 EXPONENTIAL ROUTINES

```
DEFINEF EXP(X, Y);  /* ENTRY POINT FOR THE SETL X EXP. Y. */

IF TYPE(X) EQ. INT THEN
  IF TYPE(Y) EQ. INT  THEN RETURN EXPII(X, Y);;
  IF TYPE(Y) EQ. REAL THEN RETURN EXPIR(X, Y);;
  ERRTYPE(#X EXP. Y, Y IS:#, Y);
  END IF;
IF TYPE(X) EQ. REAL THEN
  IF TYPE(Y) EQ. INT  THEN RETURN EXPRI(X, Y);;
  IF TYPE(Y) EQ. REAL THEN RETURN EXPRR(X, Y);;
  ERRTYPE(#X EXP. Y, Y IS:#, Y);
  END IF;
ERRTYPE(#X EXP. Y, X IS:#, X);

/* NOTE:  EXPIR, EXPRI, AND EXPRR ARE NOT SPECIFIED HERE. */

END EXP;



/*                          EXPII                                  */

DEFINEF EXPII(X, Y);

/* THIS FUNCTION CALCULATES X EXP. Y FOR X AND Y INTEGERS.
THE DOMAIN OF DEFINITION IS INDICATED BELOW, WHERE N > 1.

          ---------------------------
          ! ERR   +1   +1    +1   +1    !   0   E
          !                            !       X
          !  0    +1   -1    +N   -N    ! +1   P
          !                            !       O
          ! ERR   +1   -1    ERR  ERR   ! -1   N (Y)
          !                            !       E
          !  0    +1   +-1   COMPUTED   ! +N   N
          !                            !       T
          ! ERR   +1   +-1   ERR  ERR   ! -N
          ---------------------------
             0    +1   -1    +N   -N
                    BASE  (X)                  */
```

```
      VX = VALUE(X);
      VY = VALUE(Y);
      IF ABS. VX LE. 1 THEN     /* COLUMNS 1 - 3 OF TABLE. */
        IF VX EQ. 0 THEN        /* COLUMN 1. */
          IF VY LE. 0 THEN ERRVAL(≠0 EXP. N, N ≤ 0;≠, Y);;
          ELSE RETURN <INT, 0>;;     /* 0 EXP. N, N ≥ 1. */
        ELSE                    /* COLUMN 2 OR 3. */
          MINUS = VX LT. 0 AND. (VY//2) EQ. 1;
          RETURN <INT, IF MINUS THEN -1 ELSE +1>;
          END IF VX;
        END IF ABS. VX;

      /* COLUMN 4 OR 5 APPLIES. */

      IF VY LT. 0 THEN ERRVAL(≠X EXP. Y, ABS. X ≥ 2, Y < 0;≠, Y);;

      /* AT THIS POINT ABS. X ≥ 2 AND Y ≥ 0. */

      IF VY GE. (2 EXP. WS) THEN    /* WS = WORD SIZE OF COMPUTER.*/
        ERRVAL(≠X EXP. Y, ABS.X ≥ 2, RESULT WOULD NOT FIT IN ≠
             + ≠MEMORY, Y =≠, Y);;

      /* WE NOW KNOW THAT VY FITS IN ONE WORD AND HENCE VY MAY BE
      EASILY DEALT WITH IN THE LOOP BELOW. */

      R = 1;                    /* INITIALIZE RESULT. */
      XPOW2 = VX;               /* THIS IS X**1, X**2, X**4, ETC. */
LOOP:   IF (VY//2) EQ. 1 THEN R = R*XPOW2;;
        VY = VY/2;
        IF VY EQ. 0 THEN RETURN <INT, R>;;
        XPOW2 = XPOW2*XPOW2;
        GO TO LOOP;
      END EXPII;
```

```
/*                          TUPADD1                              */

DEFINE TUPADD1(T,X);

/* THIS SUBROUTINE IS EQUIVALENT TO T = T + <X>, EXCEPT THAT
IF X IS UNDEFINED, THE RESULTING TUPLE T WILL HAVE A TRAILING
UNDEFINED COMPONENT (WHICH IS AN INVALID SETL OBJECT).
     THE ARGUMENTS ARE NOT VALIDATED.  THE CONCATENATION IS
DONE IN-PLACE IF POSSIBLE. */

NEWLENGTH = NCOMPS(T) + 1;
IF NEWLENGTH GT SPACE(T) THEN
  /* NO MORE ROOM IN T. */
  T = <TUPLE, <NEWLENGTH, NCOMPALLO(NEWLENGTH), TUP(T)>>;
  END IF;
/* NOW DO THE CONCATENATION, IN PLACE. */
(TUP(T))(NEWLENGTH) = X;
NCOMPS(T) = NEWLENGTH;
RETURN;
END TUPADD1;
```

Bibliography

[1] P. Cohen. Set Theory and the Continuum Hypothesis. Benjamin Publishers, New York (1966).

[2] G. Goos. Programming Languages as a Tool in Writing System Software. In: Advanced Course on Software Engineering, Ed. Beekmann, Goos, and Künzi, Springer-Verlag, Berlin and New York (1973).

[3] J. B. Morris. A Comparison of MADCAP and SETL. Los Alamos Scientific Laboratory, University of California, Los Alamos, New Mexico (1973).

[4] J. Schwartz. On Programming. An Interim Report on the SETL Project. Installment 1: Generalities. Computer Science Department, Courant Institute of Mathematical Sciences, New York University (Feb. 1973).

[5] J. Schwartz. Set Theory as a Language for Program Specification and Programming. Computer Science Department, Courant Institute of Mathematical Sciences, New York University (1970).

Appendix 1. CATALOG OF SETL NEWSLETTERS AS OF JUNE 1975

1. BALM-SETL -- a simple implementation of SETL. M. C. Harrison
2. No longer available.
3. Modifications and extensions for SETL, Part 1.
4. APL Version of Peter Markstein's McKeeman Table Routine.
5. Miscellaneous algorithms written in SETL. J. T. Schwartz
6. Revised SETL Version of McKeeman Parse. Peter Markstein.
7. Modifications and extensions for SETL, Part 2. Dave Shields
8. Additional Miscellaneous SETL. Algorithms. J. T. Schwartz
9. Implementation and Language Design. M. C. Harrison
10. Sorting Algorithm. Kurt Maly
11. Modifications and extensions for SETL, Part 3. Dave Shields.
12. Recapitulation of the Basic Parts of the SETL Language. J. Schwart
13. Additional miscellaneous algorithms. J. T. Schwartz
14. Additional syntactic extensions. J. T. Schwartz
15. A proposed SETL implementation plan through the end of the bootstrap phase. J. T. Schwartz
16. SETL 64-character set -- 48-character set/ 026 keypuncher -- CDC 6600 64-character set / 029 keypuncher. Kurt Maly
17. No longer available.
18. Preliminary specification of BALMSETL conventions. D. Shields
19. Lexical Description of SETL. Kurt Maly
20. BALMSETL User's Guide (in brief). D. Shields
21. An Outside Review: Comments on the SETL Draft.
22. Some small and large language extensions for consideration. J. Schwartz
23. Current Status of BALMSETL Implementation. D. Shields
24. Description of a Register Allocation Algorithm. K. Kennedy
25. Print Routine. B. Loerinc
26. The currently specified form of SETL from a more fundamental point of view. J. T. Schwartz
27. Code for the Postparse Setup Procedure (Postparse metalanguage analysis). J. T. Schwartz
28. An Algorithm for Common Subexpression Elimination and Code Motion. K. Kennedy

29. Some issues connected with subroutine linkage. J. T. Schwartz
30. Sinister calls. J. T. Schwartz
31. An additional preliminary remark on the importance of "object types" for SETL, with some reflections on the motion of "data structure language". J. T. Schwartz
32. Hyper-SETL procedural languages. J. T. Schwartz
33. What is Programming? J. T. Schwartz
34. Syntax revisions in preparation for implementation. J. Schwartz
35. New form for IFF-statement. Dave Shields
36. Syntactic and semantic conventions for programmer-definable object types (not available; replaced by Newsletter 76) JTS
37. Initial description of an algorithm for use-definition chaining in optimization. P. Owens, K. Kennedy.
38. Algorithm for live-dead analysis including node-splitting for irreducible program graphs. K. Kennedy.
39. More detailed suggestions concerning 'data strategy' elaborations for SETL.
40. List of Newsletters 1-40.
41. Additional planning detail for the current and next phase of SETL implementation. J. T. Schwartz
42. Revised conventions concerning tuples. J. T. Schwartz
43. A parsing scheme for FORTRAN. S. Gruber
44. Comprehensive SETL specifications. K. Maly
44a Modifications to Newsletter 44. K. Maly
45. Semi-local SETL optimization. D. Shields
46. Generalized nodal span parse routine; preliminary draft. J.T.S.
47. Outline for a parsing scheme for SETL. K. Maly
48. Toward a documentation of the String project's program for parsing English sentences. J. Hobbs.
49. Detailed specifications of certain SETL operations. H. Warren
50. Three-phase parsing scheme for SETL. K. Maly
51. List of Newsletters 41-51.
52. Comments on SETL. Jay Earley
53. SETL to LITTLE Translation: a first look. H. S. Warren
54. Current status of BALM-SETL 4. S. Gruber.
SETL suggestions and questions. S. Finkelstein.
56. Additional comments on some basic SETL operations. J. Earley.

56a. More Comments on SETL. J. Earley

56b. More SETL Comments. J. Earley

57. Minimizing Copying in SETL: Preliminary Observations. H. Warren

58. Phase One of the SETL Compiler. K. Maly

59. An Algebra of Assignment. R. A. Krutar

59a. An Algebra of Assignment - Addenda and Errata. R. Krutar

60. SETL Compiled Code: Calls to SETL Procedures. H. Warren

61. Syntactic Structure of SETL. Kurt Maly

62. Final Specification Part of SETL and Parser. Kurt Maly

63. The SETL Print Routine. G. Fisher

64. SETL Compiler with Elaborated Data Structures. K. Maly

65. Some Notational Suggestions. Robert Bonic

66. BALMSETL Users Manual Version 1.0. Ellie Milgrom

67. Data Structures of the SETL Compiler from the LITTLE Version. Kurt Maly

68. Some Thoughts on Efficient Programming in SETLB. S. Brown

69. List of Newsletters 52-86.

70. SETL Users Manual. J. T. Schwartz

71. Deducing the Logical Structure of Objects Occurring in SETL Programs. J. T. Schwartz

72. An Introductory Explanation of SETL. A Status Review and Profile of SETL User-Group. D. Shields

73. USER'S Guide to SETL Run-Time Library. K. Maly and H. Warren

74. Project Plan for First Stage of Implementation. L. V. Gorodnaya, D. A. Levin, V. Chernobrod

75. Some Tboughts on the Use of BALM to Implement SETL. E. Milgrom [also: this is BALM Bulletin No. 13]

76. Semantic-Definition Matters. J. T. Schwartz and G. Jennings

77. Transferring SETLB to Other Machines. J. T. Schwartz

78. Executing BALM and SETLB at NYU Courant. R. Paige

79. SETL, LITTLE, BALM Files. R. Paige

80. Algorithms in the SETLB Test Package. Kent Curtis

81. Memory Size of SETLB Runs. J. T. Schwartz and S. Brown

82. Timing Comparision between SETLB and FORTRAN. E. Desautels

83. User Experience and Human Factors. J. T. Schwartz

84. Plan for a Library of Algorithms. J. T. Schwartz

85. Preliminary estimate of minimum running sizes for the next SETL system. H. Warren and J. Schwartz
86. Proposal for a temporary, but easily implemented, software paging scheme. J. Schwartz.
87. Workplan for the next phase of SETL implementation. J. Schwartz.
88. A scheme for BALMSETL measurements. J. Schwartz
89. User information for lexical scan setup package. E. Guth and B.
90. Preliminary reflections on the use of SETL in a data base context. J. Schwartz.
91. A grammarless parse and a related method of retrieval by similarity. J. Schwartz
92. Some experiments with SETLB programs. K. Curtis.
93. A note on optimization and programming style in SETL. K. Curtis.
94. An algorithm to represent a collection of sets as intervals (on a line). G. Jennings.
95. Generalized nodal span parse routine; corrected version. Y. Feinroth.
96. Pointers and 'very high level languages'. N. Minsky.
97. SETL extensions for operating system description. P. Markstein.
98. Reflections on P. Markstein's newsletter on SETL extensions for operating system description. J. Schwartz.
99. Paging, the quick and dirty way (also BALM Bulletin 21). S. Brown.
100. Making SRTL debugging runs. H. Warren.
101. How to program if you must (the SETL style). R. Bonic.
102. Reduction in strength using hashed temporaries. K. Kennedy.
103. Preliminary plan for BALM-to-LITTLE translator. J. Schwartz.
104. An algorithm to represent a collection of sets as a direct product of intervals on the line. G. Jennings.
105. A SETL program for basic block optimization. S. Marateck.
106. User variation of the semantics of function and subroutine invocation. G. Jennings.
107. Linear function test replacement. K. Kennedy.

108. APL-SETL, An extension of SETL achieved from user varied semantics. G. Jennings.
109. Faster execution for the LITTLE-based BALM system. S. Brown.
110. More on semantic definition matters. J. Schwartz.
111. Global dead computation elimination. K. Kennedy.
112. An algorithm to compute use-definition chains. K. Kennedy.
113. LITTLE code generation from the BALM compiler. S. Brown.
114. A SETLB to publication SETL translator. A. Getzler.
115. A SETL representation of the Maryland GRAAL graph manipulation language. G. Weinberger and A. Tenenbaum.
116. Catalog of SETL(C) newsletters as of July 30, 1973. D. Y. Levin, L. V. Chernobrod (Novosibirsk).
117. A static debugging system for LITTLE. E. Schonberg.
118. Revised and extended algorithms for deducing the types of objects occurring in SETL programs. A. Tenenbaum.
119. A suggested generalization and revision of the SETL compound operator form. J. Schwartz.
120. A general-recursive extension of functional application and its uses. J. Schwartz.
121. An algorithm to determine the identity of SETL run-time objects. A. Tenenbaum.
122. More local and semilocal SETL optimizations. J. Schwartz
122A A few peephole optimizations applicable to iterators. J. Schwartz
122B Still more miscellaneous optimizations. J. Schwartz
123. Variable subsumption with constant folding. K. Kennedy
124. The VERS2 language of J. Earley considered in relation to SETL. E. Schonberg.
125. Schaefer's node splitting algorithm. K. Kennedy.
126. A SETLB specification of EDIT. M. Brenner.
127. Edge-listing data-flow algorithms. K. Kennedy.
128. A LITTLE written translator from SETL to LITTLE. E. Schonberg and S. Brown.
129. More on SETL in a data-base environment. J. Schwartz.
130. Deducing relationships of inclusion and membership in SETL programs. J. Schwartz.
130A Estimates from below the domain of a mapping. J. Schwartz.
130B The use of equalities in the deduction of inclusion/membership relations. J. Schwartz.

131. More on copy optimization of SETL programs. J. Schwartz.

132. Some optimizations using type information. J. Schwartz.

3. A higher level control diction. J. Schwartz.

133A Additional pursue block examples. J. Schwartz.

133B General comments on high level dictions, and specific suggestions concerning 'converge' iterators and some related dictions. J. Schwartz.

134. Interprocedural optimization. J. Schwartz.

135. Introductory lecture at the June 28 'informal optimization symposium'. J. Schwartz.

135A Structureless programming, or the notion of 'rubble' and the reduction of programs to rubble. J. Schwartz.

135B Additional thoughts on 'language level' and optimization. J. Schwartz.

136. A framework for certain kinds of high-level optimization. J. Schwartz.

137. Additional thoughts concerning automatic data structure choice. J. Schwartz.

138. On J. Earley's 'method of iterator inversion'. J. Schwartz and R. Paige.

138A Optimization by set suppression. J. Schwartz.

140. Use-use chaining as a technique in typefinding. J. Schwartz.

141. Reflections on some very high level dictions having an English/ 'automatic programming' flavor. J. Schwartz.

142. What programmers should know. J. Schwartz.

143. 'Arguments from use' in the proof of relationships of inclusion and membership. J. Schwartz.

144. Interprocedural live-dead analysis. J. Schwartz.

145. GYVE-oriented inter-process coordination and control features for an extended SETL (SETLG). J. Schwartz.

146. Adaptation of GYVE/SETLG to distributed networks of computers. J. Schwartz.

147. A syntactic construct useful for checking parameters. J. Schwartz

148. Technical and human factors improvements for the fully compiled SETL system. E. Schonberg and A. Stein.

149. Conventions allowing other languages to be used within GYVE; files; memory hierarchy questions; some suggestions for GYVE extensions. J. Schwartz.

150. What constitutes progress in programming? J. Schwartz.

Index

In addition to items of general interest, the following index lists all named SETL primitive operators, subroutines, and functions defined in the present volume. These 'procedure names' are shown in italics. In accordance with general SETL conventions, the names of infix and prefix operators are underlined.