Courant Institute of
Mathematical Sciences

A SETLB Primer

Henry Mullish and Max Goldstein

New York University

A SETLB Primer

(With over 100 illustrative programettes)

Henry Mullish and Max Goldstein

1973

Courant Institute of Mathematical Sciences
New York University

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

		Page
1.	Introduction	1
1.1	The SETLB Character Set	3
1.2	SETLB Programs	4
1.3	Control Cards	5
1.4	Elementary Programs; the HELP Debugging Aid; Comments in Programs	6
	Questions	10
2.	Sets	11
2.1	Explicit Set Formers	11
2.2	Elementary Operations on Sets	13
2.3	Elements vs. Subsets	17
2.4	Comparison and Boolean Operators	20
2.5	More on Subsets	23
2.6	Union of Sets	27
2.7	Symmetric Difference	28
2.8	The Number of Elements in a Set	29
2.9	Duplicate Elements	30
2.10	Changing the Contents of Sets	31
2.11	The "Selection or "Arbitrary Element" Operator	34
	Questions	36
3.	Tuples	39
3.1	Indexing of Tuples	41
3.2	The Zeroth Component of a Tuple: An Illustration of Error Termination	42
3.3	Modifying Tuples	44
3.4	The 'Undefined Value' or 'Omega' Concept; Additional Remarks on Error Termination	48
3.5	The Head and Tail Operators	50
	Questions	51
4.	Additional Information on Sets and Tuples	52
4.1	Set Formers	52
4.2	A Short Digression on Arithmetic in SETLB	56
4.3	More Examples of Set-Formers	57
4.4	Existentials	59
4.5	Universal Quantifier	62

4.6	'Multiple' Assignments	67
	Questions	70
5.	Sets of Pairs and Tuples Used as Maps	73
5.1	Sets of Tuples as Functions	73
5.2	An Observation on the Use of Subexpressions within Set Expressions	83
	Questions	84
6.	Control Statements	86
6.1	Integration over a Numerical Range	86
6.2	Compound Operators	90
6.2.1	Examples of Set Formers and Compound Operator	92
6.2.1.1	A Prime Number Generator and the Sum of Primes	92
6.2.1.2	Checking a Formula	94
6.3	Iteration over the Elements of a Set	95
6.3	Other Iteration Forms	99
6.4	IF, THEN, ELSE	108
6.5	The WHILE Iterator	110
6.6	Labels and GO-TO Statements	114
6.7	A Remark on Programming Style: GO TO -less Programming	116
6.8	Conditional Expressions	117
	Questions	120
7.	Character Strings	122
7.1	Substrings	125
	Questions	126
8.	More Examples of the Use of SETLB	127
8.1	A Sorting Algorithm	127
8.2	Counting Character Frequencies	128
9.	Subprograms	130
9.1	User-Defined Functions	1 0
9.2	Subroutines	132
	Questions	137
10.	Built-In Functions and Operators Provided by SETLB	138
10.1	Absolute Value Operator	138
10.2	The Maximum and the Minimum Operators	139
10.3	The Random Function	141
10.4	IN. and OUT.	142

10.5	The FROM. Operator	144
10.6	The NOOP Instruction	145
10.7	The IS., or General Assignment to the Right, Operator	146
10.8	NEWAT.	147
10.9	Object Types	148
10.10	The ASSERT Debug-Print Statement	150
10.11	Macros	151
10.12	User-Defined Binary and Monadic Operators	154
	Questions	157
11.	Reading from Data Cards	160
12.	Some Sample Programs	164
12.1	A First Full-Scale Example of the Use of SETLB: The Koenigsberg Bridge Problem	164
12.2	A Second Full-Scale Example of the Use of SETLB: Translating to Pig Latin	169
13.	Summary of SETLB Features	176
14.	Miscellaneous Advanced Information	184
15.	Bewares	198
	Related literature	199
	Index	200

1. Introduction

Most of this introductory section is intended for the person who has had experience with computer languages of various kinds. In it we will try to describe the general relationship of SETLB to other computer languages. The novice will surely find the points made somewhat obscure. But this should not in any way deter the non-sophisticate from going on with the primer to learn "what SETLB is all about." In most of the primer not even a knowledge of set theory is assumed; all required information is explained in the body of the text, hopefully in such a manner that most, if not all, of the material can be absorbed at the first reading. Novices to programming are advised to skip this introduction and to begin immediately with Section 1.1.

SETL is a new programming language whose essential features are taken from the mathematical theory of sets. SETL has a precisely defined formal syntax as well as a semantic interpretation and thus it permits one to write programs for compilation and execution.

The SETL language was first described in a manuscript entitled "Abstract Algorithms and a Set-Theoretic Language for Their Expression," by Jacob T. Schwartz. In the preface and introduction to this manuscript Prof. Schwartz discusses the relationship between mathematics and programming and stresses the need for a new very high level language for the specification of complex algorithms. A mathematicized programming set theoretic language is seen as answering this need. A later version of this same material will be found in "On Programming: An Interim Report on the SETL Project," by the same author (Courant Institute Lecture Notes).

Having general finite sets as its fundamental objects, SETL is a language of very high level, i.e., the language incorporates complex structured data objects and permits global operations upon them, thus freeing the user from the onerous task of specifying the detailed internal forms which are to represent the structured objects. That is, SETL allows one to specify even very complicated algorithms without regard to the details of possible data structures. It relieves the user of details concerning layout of arrays, use of pointers, etc. -- details which are inevitably encountered early in the treatment of a complex programming problem by conventional techniques. It therefore permits one to concentrate on the logical structure of the projected program. The programmer is thereby freed to describe abstract problem-related entities and their interactions in a familiar and analytically natural manner.

Of course, one pays a price for these great advantages. Namely, it becomes all too easy to generate very inefficient programs. Generally speaking, SETL pays a substantial price in efficiency for its logical power. Nevertheless, it is our feeling that SETL will be useful in a variety of significant ways. It is a language in which complex algorithmic processes can be formally and precisely defined.

SETL may become quite useful in the teaching of computer science. SETL-based introductory courses could treat abstract algorithms separately from concrete algorithms for which data structures have been specified. SETL allows complex algorithmic processes to be represented and analyzed independently of the way in which the logical objects to which they refer are mapped onto a computer.

The subset verion of SETL which is currently implemented on the CDC 6600 at the Courant Institute is called SETLB. The suffix "B" indicates that the current version was written using the extendable, LISP-like language called BALM, conceived and implemented by Professor Malcolm Harrison, also of the Courant Institute.

SETLB is implemented using a preprocessor to "BALMSETL"; BALMSETL is an extension of BALM obeying all the syntactic and semantic conventions of BALM. When full SETL (rather than SETLB) is implemented, many of the current implementation-related restrictions imposed by the presently existing linkage to the BALM language will be alleviated. Even though SETL will then differ in some important respects from the present SETLB, a user of the current version of SETLB will find that his programs require only slight modification.

We wish to acknowledge the assistance of the SETLB group, in particular Prof. Jacob T. Schwartz, Kent Curtis, Robert Bonic, Dave Shields, Hank Warren, and Steve Tihor, who not only answered our questions with patience, but contributed algorithms and reviewed the manuscript.

The work reported in the text is supported by NSF Grant NSF-GP-1202X and under AEC Contract AT(11-1)-3077.

What follows is an account of the SETLB language.

1.1. The SETLB Character Set

At Courant Institute there are several modified 029 keypunch machines. The characters used in SETLB are immediately obtainable on the 029 keypunch whereas some of them must be multipunched on a 026 keypunch.

All of the 029 characters are used in SETLB, with the sole exception of the ¬ (not) sign. The SETLB characters are the 26 letters of the alphabet, the 10 digits of the decimal system, and the special characters which are illustrated below, together with their punched hole representation for the CDC 6600.

=	,	$	.	≠	(	*	)	-	/	¬	→	↑	<
3	0	11	12	4	0	11	12	11	0	12	5	11	12
8	3	3	3	8	4	4	4		1	6	8	5	2
	8	8	8		8	8	8			8		8	8

]	;	+	≡	:	↓	>	≤	∧	[	%	∨	≥
0	12	12	0	2	11	11	5	0	7	6	11	12
2	7		6	8	6	7	8	7	8	8	2	5
8	8		8		8	8		8			8	8

Any of the above synbols may be punched on the 026 by depressing the MULTipunch key while individually punching each hold. Despite the peculiar appearance of the resulting composite print on the card, it will be acceptable on the computer and that is the primary consideration.

1.2. SETLB Programs

SETLB programs are punched on cards in columns 1 through 72. However, in view of the possibility that some time in the future column 1 may be reserved for special purposes it is recommended that the instructions occupy columns 2 through 72. There are no fixed fields such as in Fortran, etc. Each statement must be terminated by a semicolon.

All programs must be structured into blocks which commence with DO; and terminate with COMPUTE;. Furthermore, every SETLB program must end with a FINISH; statement.

The division of a SETLB program into several DO;...;COMPUTE; blocks aids in debugging SETLB programs for the following reason: If an error occurs during the execution of any one block the entire program will not be aborted but rather only the one block containing the error. Subsequent blocks will be executed "without prejudice," as it were.

In order to print a value one uses the PRINT. instruction. The keyword PRINT. is followed by the list of expressions that are to be printed. For example:

(a) PRINT. A;

(b) PRINT. A,B;

(c) PRINT. C, C+D, E*F;

For our first sample SETLB program we shall set A=10, B=3 and C=2. We then print out A,B,C, A+B, A+C, B+C, A*C, B/C, A-C and review the output produced.

The complete program is:

```
DO;
   A=10;
   B=3;
   C=2;
   PRINT. A,B,C,A+B,A+C,B+C,A*C,B/C,A-C;
COMPUTE;
FINISH;
```

1.3 Control Cards

Before showing the form of the resulting output we advise the would-be SETLB user that, as usual, certain control cards must be included in order for even the simplest program to be accepted by the system. We shall now show the complete, somewhat intimidating, set of control cards without attempting any explanation of them. Here is the complete physical deck for the above program as presented to the CDC 6600.

```
ID,DT30,CM200000.MULLISH
ATTACH,SETLABS,SETLB.
LOADER.RFL,66000.
SETLABS.(HELP)
ATTACH(BLM4SVD,SAVESETL)
ATTACH,BALMTR,BALMTRANS.
RFL,200000.
BALMTR,SETLOUT.
E-O-R       (end-of-record card)
    DO;
        A=10;
        B=3;
        C=2;
        PRINT. A,B,C,A+B,A+C,B+C,A*C,B/C,A-C;
    COMPUTE;
    FINISH;
E-O-F       (end-of-file card)
```

1.4 Elementary Programs; the HELP Debugging Aid; Comments in Programs

Here is the output of the above SETLB program, preceded by a printed listing of the program. Each line of the program is sequentially numbered by numbers placed under the heading LINE NO. Other auxiliary statement numbers are printed under the heading STATE NO. Since SETLB at present handles only integer arithmetic, it is interesting to note the value obtained for B/C.

Program 1

```
LINE STATE
 NO    NO

  1     0     /* AN ELEMENTARY SETLB PROGRAM, USING -HELP- */
  2     0     DO;
  3     0         A=10;
  4     1         B=3;
  5     2         C=2;
  6     3     PRINT,A,B,C,A+B,A+C,B+C,A*C,B/C,A-C;
  7     4     COMPUTE;
  8     4     FINISH;
```

Output -- Program 1

```
- - - AT 1 IN MAIN A IS 10
- - - AT 2 IN MAIN B IS 3
- - - AT 3 IN MAIN C IS 2
10 3 2 13 12 5 20 1 8

* * * (END OF FILE ON INPUT) * * *
```

The first thing to be noticed in the above output are the first three output lines, each of which is preceded by --- . These lines are produced by a SETL debugging feature which prints out the values of all variables to which assignments have been made. This feature was activated by the inclusion of the word (HELP) on the fourth control card listed above. The user will find that this (HELP) feature is generally of extreme utility in debugging programs. We mention it here merely to alert the user to its existence. This important debugging feature is discussed in detail in Section 14. In describing programs and their output subsequently, (HELP) will be left switched off.

The proper (as distinct from debug) output from the program appears on the fourth output line. The first three values are those for A, B and C, namely 10, 3 and 2. The value of A+B is seen to be 13, A+C is 12, B+C is 5, A*C is 20, B/C, i.e. 3/2, appears as the <u>truncated value</u> 1; finally, the value of A-C is 8. Before passing on from this program it is worth pointing out that in SETLB it is quite proper to include unevaluated expressions in a print list, even though this is a somewhat uncommon feature in programming languages. We also note that the list of results is printed on a single line and separated by a single space, in conformity with the use of a single print instruction.

The same program was run again, this time with the (HELP) feature omitted; here is the listing and printed output. Note that with the (HELP) feature omitted no numbers appear under the heading STATE NO.

Program 2

```
LINE STATE
 NO    NO

        /* THE SAME ELEMENTARY PROGRAM, WITHOUT USING *HELP* */
        DO;
             A=10;
             B=3;
             C=2;
        PRINT,A,B,C,A+B,A+C,B+C,A*C,B/C,A-C;
        COMPUTE;
        FINISH;
```

Output -- Program 2

```
10 3 2 13 12 5 20 1 8

* * * (END OF FILE ON INPUT) * * *
```

To the above program and output one may make the objection that in order to understand the output one has to refer directly to the program. Obviously, this is not the most satisfactory way of printing output. It is a good idea to print some description or other prior to each number outputted in order to identify it. SETLB makes it easy to print fixed text. Text to be printed must be enclosed by quote signs. On the 029 keypunch machine the quote sign is the ≠ character.

Another way to improve the readability of a program is to include explanatory comments. This may be done by using the $ symbol. Anything punched on a card followed by the $ symbol is ignored by the computer although it will appear in a listing of the program.

Program 3

```
LINE STATE
 NO   NO

$ PROGRAM 3
$
$   THIS IS JUST A SLIGHT MODIFICATION TO THE PREVIOUS PROGRAM
$
$   (HELP) HAS BEEN OMITTED FROM THE CONTROL CARDS
 DO;
      A=10;
      B=3;
      C=2;
 PRINT,≠A=≠,A,≠B=≠,B,≠C=≠,C,≠A+B=≠,A+B,≠A+C=≠,A+C,≠B+C=≠,B+C,≠A*C=≠,A*C
 ,≠B/C=≠,B/C,≠A-C=≠,A-C;
 COMPUTE;
 FINISH;
```

Output -- Program 3

```
≠A=≠ 10 ≠B=≠ 3 ≠C=≠ 2 ≠A+B=≠ 13 ≠A+C=≠ 12 ≠B+C=≠ 5 ≠A*C=≠ 20 ≠B/C=≠
1 ≠A-C=≠ 8

* * * (END OF FILE ON INPUT) * * *
```

It will be noticed in the above that the print statement is now too long to be contained on one card. (Remember one must not punch beyond column 72.) All one has to do in such a situation is to continue punching the instruction on another card <u>without any special preliminaries</u>. This process may be continued indefinitely. A statement is always terminated by the occurrence of a semicolon.

Another form of comment card, resembling that used in the PL/1 programming language is available in SETLB. This second form of comment begins with /* and terminates with */. Of course, this convention obviates the possibility of including a */ in a comment, but this is no great hardship.

QUESTIONS

Chapter 1

1. Why are the following SETLB programs invalid:

(a)
```
A=1; B=2; C=3;
PRINT. A,B,C
COMPUTE; FINISH;
```

(b)
```
DO;
X=1/2; Y=3/2; Z=4/2;
PRINT. X,Y,Z
KOMPUTE: FINISH;
```

A separate booklet containing the answers to these and all the other questions included in this primer is obtainable by writing on official stationery to the authors.

2. Sets

The SETL language aims to make it possible to state problems in the language of set theory. A set in SETLB is a clearly defined finite collection of elements.

2.1 Explicit Set Formers

A set can be formed by the explicit enumeration of its elements as in:

A = {5,1,7,2}

The order in which the elements appears is of no consequence. In other words the set

B = {2,1,5,7}

is identical to the set A above.

A set may have another set, as one of its elements.

C = {'FUN', {5,1,7,2}}

Here, C is a set having two elements, the first a character string, the second a set of numbers.

By definition, a set has no duplicate elements. Hence, the statement

D = {5,5,1,7,7,2}

assigns D the same value as set A or B above.

One quite useful set contains no elements whatever. This special set is called the null set. It is denoted by { } or Ø in set theory and by NL. in SETLB. Note that the set {0} is not the null set -- it is a set with the number zero as its single element.

Since the characters { } are not available on the 029 keypunch, explicitly enumerated sets are represented in SETLB as follows: A = ≤:5,1,7,2≥; . That is, we begin the enumeration of a set with the ≤ character, followed immediately by a colon, and then give the elements of the set separated by commas. The enumeration is terminated by the ≥ symbol.

The next program simply specifies several sets in SETLB notation and prints them. Its output merits some discussion.

Program 4

```
LINE STATE
 NO   NO

         /* THESE ARE SOME EXAMPLES OF SETS */
        $
         DO;
         A=≤:5,1,7,2≥;
         B=≤:-1,-5,-3,-4≥;
         C=≤:≠JACK≠,≠DAVE≠,≠MAX≠,≠HENRY≠≥;
         D=≤:A,B,C≥;
         PRINT,≠A=≠,A,≠B=≠,B;
         PRINT,≠C=≠,C;
         PRINT,≠D=≠,D;
         E=≤:1,1,2,2,3≥;
         PRINT,≠E=≠,E;
         F=≤:1 1 2 2 3≥;
         PRINT,≠F=≠,F;
         COMPUTE; FINISH;
```

Output -- Program 4

```
≠A=≠ ≤1 2 5 7≥ ≠B=≠ ≤-1 -3 -4 -5≥
≠C=≠ ≤≠HENRY≠ ≠MAX≠ ≠JACK≠ ≠DAVE≠≥
≠D=≠ ≤≤1 2 5 7≥ ≤-1 -3 -4 -5≥ ≤≠HENRY≠ ≠MAX≠ ≠JACK≠ ≠DAVE≠≥≥
≠E=≠ ≤1 2 3≥
≠F=≠ ≤11223≥

* * * (END OF FILE ON INPUT) * * *
```

Line #8 of the above program instructs the computer to print out sets A and B. Both these sets will be printed on a single line. Notice that the elements are not printed in the same order as that in which the set was initially enumerated. (They are in fact printed in the order in which they are maintained internally within the computer.) Moreover, there are no commas separating the elements printed, but rather blank spaces. Nor is the colon or the terminal semicolon printed.

Line #9 prints out the set C and once again we notice that the order of printing is arbitrary and that blanks separate the elements.

Line #10 instructs the compuer to print out the set D. This set has A, B and C, which are themselves sets, as its elements. Hence the printout beginning with $\leq\leq$ and concluding with $\geq\geq$, which at first sight seems somewhat peculiar.

Line #11 enumerates the set E in a manner involving duplicate elements. The printout shows clearly that duplicates are ignored.

Finally, line #14 prints out the set F. Look carefully at the definition of F! At first sight it may look much like the definition of the set E; however, instead of commas being used to separate elements, spaces are used. These spaces, however, are ignored by the SETLB compiler and as a result and set F has a single element, namely 11223.

2.2 Elementary Operations on Sets

Given a set A one can ask if a particular element x is a member of that set. In set-theoretic notation, this relationship is written

(1) $$x \in A$$

Or we can ask whether x is <u>not</u> an element of A

(2) $$x \notin A$$

The answer to such a function is either 'true' or 'false'.

By the same token we can ask if a set A is equal to a set B, or whether A is not equal to B. Lastly, one may want to know whether a set A is a subset of set B.

To all of these questions the answer is either 'yes' or 'no', 'true' or false'. Expressions like (1) or (2) are therefore called Boolean <u>valued</u> expressions. Boolean values, once formed, may be combined using the logical operators "and" and "or", care being taken to parenthesize so as to show the intended groupings.

In the next program a set A is defined.

$$A = \underline{<}:5,1,7,2\underline{>};$$

The question is then asked: is 3 an element of the set A? In SETLB this is done with the → sign, used because the conventional mathematical sign '∈' is not available on the 029 keypunch.

$$3 \rightarrow A$$

The question "Is 7 an element of A?" is written

$$7 \rightarrow A$$

Finally, one asks: is 10 an element of A? This is written

$$10 \rightarrow A\ .$$

Plainly, 3 and 10 are not elements of A while 7 is. Since these three tests appear on the same PRINT. statement, the three answers

FALSE TRUE FALSE

appear on the same line.

Next, the question is asked whether 3 is an element of A, but the result of this test is inverted using the logical operator NOT. Since 3 is not an element of A, 3 → A is FALSE. The <u>NOT</u>. converts the result FALSE to TRUE which is printed. Similarly, for the remaining two expressions.

Program 5

```
LINE STATE
 NO    NO

/* SOME BOOLEAN OPERATIONS - ELEMENT TESTS */
DO;
A=≤:5,1,7,2≥;
PRINT. 3→A,7→A,10→A;
PRINT.NOT.(3→A),NOT.(1→A),NOT,(10→A);
COMPUTE; FINISH;
```

Output -- Program 5

```
FALSE TRUE FALSE
TRUE FALSE TRUE

* * * (END OF FILE ON INPUT) * * *
```

Three elementary set relationships are tested in the next program; each test asks whether a given element is a member of a specific set. Since each expression occurs in a separate PRINT. instruction, the answers occupy separate lines.

Program 6

```
LINE STATE
 NO   NO

          /* SOME MORE ELEMENT TESTS */
          DO;
          PRINT. 3→≤;5,6≥;
          PRINT.2→≤:1,5,8,2≥;
          PRINT.1→≤:2,4,6≥;
          COMPUTE; FINISH;
```

Output -- Program 6

```
FALSE
TRUE
FALSE

* * * (END OF FILE ON INPUT) * * *
```

2.3 Elements vs. Subsets

Suppose we define a set by

A = ≤:3,4, ≤:1,2≥≥;

This is a set composed of three elements, namely 3,4 and an element which itself is the set ≤:1,2≥. Sets as set-elements are valid both in mathematics and in SETLB. We can then ask whether the set ≤:4,3≥ is an element of A. In other words, is one of the elements in the set A identical with ≤:4,3≥ ? The answer is obviously negative and so we expect a FALSE printout.

Inserting the logical operator NOT. before this test changes the truth value obtained from FALSE to TRUE. This is seen in the following printout.

Program 7

```
LINE STATE
 NO   NO

          /* SOME SUBSET TESTS */
          DO;
          PRINT.≤:4,3≥→≤:3,4,≤:1,2≥≥;
          PRINT.NOT.≤:4,3≥→≤:3,4,≤:1,2≥≥;
          COMPUTE; FINISH;
```

Output -- Program 7

```
FALSE
TRUE

* * * (END OF FILE ON INPUT) * * *
```

In dealing with sets, as distinct from other types of compound SETLB objects to be discussed later, the order in which the elements appear is of no consequence. The set

A = ≤:1,2,3≥;

is exactly equal to:

B = ≤:3,1,2≥;

in which the same elements are enumerated in a different order. It therefore follows that the sets A and B above are equal. This can be verified using the EQ. operator. The sets

C = ≤:8≥; and D = ≤:9≥;

are not equal; this can be verified using the NE. (not equal) operator.

If we define the two following sets:

E = ≤:3,4≥;

F = ≤:1,2,3,4≥;

can ask whether the set F has the set E as a subset. This test can be made using the INCS. (set inclusion) operator. Since E is a subset of F, F includes E, and the Boolean expression

F INCS. E

yields the result TRUE.

All this is shown in the program which follows.

Program 8

```
LINE STATE
 NO   NO

          /* SET EQUALITY; SUBSETS */
          DO;
             A=≤:1,2,3≥;
             B=≤:3,1,2≥;
          PRINT.(A EQ,B);
             C=≤:8≥;
             D=≤:9≥;
          PRINT.(C NE,D);
             E=≤:3,4≥;
             F=≤:1,2,3,4≥;
          PRINT.(F INCS,E);
          COMPUTE; FINISH;
```

Output -- Program 8

```
TRUE
TRUE
TRUE

* * * (END OF FILE ON INPUT) * * *
```

2.4 Comparison and Boolean Operators

The comparison operators LT., LE., GT., GE., EQ., NE. and the Boolean operators AND., NOT. and OR. are very useful. The following program is intended as an elementary illustration of these operations.

Program 9

```
LINE STATE
 NO    NO

/* COMPARISON OPERATORS */
DO;
PRINT, 6 LT, 8 AND, 11 GT, 4;
PRINT, 6 LE, 6 AND, 11 GE, 4;
PRINT, 8 GT, 1 OR, 2 GT, 100;
PRINT, 5 GE, 1 AND, 2 GT, 2;
COMPUTE; FINISH;
```

Output -- Program 9

```
TRUE
TRUE
TRUE
FALSE

* * * (END OF FILE ON INPUT) * * *
```

The next program shows that SETLB allows the abbreviation A. to be substituted for AND. and O. for the OR. operator. The abbreviation 'N.' for the NOT. operator is also valid. Furthermore, the abbreviation 'T.' for TRUE. and 'F.' for FALSE. are also available.

Program 10

```
LINE STATE
 NO   NO

  1          /* MORE ON LOGICALS */
  2          DO; A=T.; B=F.;
  3          PRINT. A OR.B;
  4          PRINT. A O. B;
  5          PRINT. A AND. B;
  6          PRINT. A A. B;
  7          COMPUTE; FINISH;
```

Output -- Program 10

```
TRUE
TRUE
FALSE
FALSE

* * * (END OF FILE ON INPUT) * * *
```

In the following program we evaluate Boolean expressions involving a set C and combine these expressions using certain Boolean operations including the AND. and OR. operators. The reader should reason his way to the results of the program and compare them with the printed output. Notice the abundant use of parentheses, made necessary by the rather unhelpful SETLB precedence rules. In SETLB unparenthesized expressions associate to the right, contrary to convention. Thus:

$$4 \to C \text{ AND. } B$$

is seen by SETLB as

$$4 \to (C \text{ AND. } B)$$

and not as

$$(4 \to C) \text{ AND. } B \quad .$$

SETLB users should always use parentheses to indicate the desired grouping.

<u>Program 11</u>

```
LINE STATE
 NO    NO

     /* SOME BOOLEAN OPERATIONS */
     DO;
     C=≤:5,7,9,11≥;
     PRINT,(4→C) AND, (9→C);
     PRINT,(5→C) AND,(9→C);
     PRINT,(4→C) OR, (9→C);
     PRINT,(7→C) AND, ((5→C) OR,(6→C));
     COMPUTE; FINISH;
```

<u>Output -- Program 11</u>

```
FALSE
TRUE
TRUE
TRUE

* * * (END OF FILE ON INPUT) * * *
```

2.5 More on Subsets

Pay careful attention to the difference between 'subset' and 'element'. For example, examine the following:

A = ≤:1,2≥;

B = ≤:1,2,3,4≥;

Does B include the set A? That is, is A a subset of B? Since all the elements of A appear as elements in B the answer is TRUE. But now let:

C = ≤:≤:1,2≥,3,4≥;

Here we have a set C whose first element is the set containing the elements 1 and 2. Does the set C include the set A? The answer this time is that it does not. In fact 1 is not an element of C, but is rather an *element of an element* of C, which is quite a different matter. The *elements* of C are the integers 3, 4 and the set ≤:1,2≥.

In order to understand more clearly what is involved here let us write out all the subsets of B and C, producing a list which incidentally includes the sets B and C themselves. Note that the null set is a subset of every set.

Subsets of B

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

{3,4}, {2,4}, {1,4}, {1,2,4}, {2,3,4}, {1,3,4} {4}, { } .

Subsets of C

{{1,2}, 3,4}, {{1,2}}, {{1,2}, 3}, {3}, {{1,2}, 4}, {3,4}, {4}, { } .

From the above it is clear that the set {1,2} is definitely a subset of B but not of C. In answer to the question: is A an element of C, it certainly is. Is 1 an element of C? -- no, it is not.

The following program's output shows that B includes the set A while C does not include the set A. Whereas it is true that A is an element of C, 1 is not an element of C. Clearly A is not eq-al to B and it is true that B is not equal to C. Also, 1 is an element of A and 4 of B. The element 23 is not a member of A, but since 4 is a member of C the Boolean expression incorporating the OR. is true. All these possibilities are made use of in the next program, the last expression of which is a combination of an AND. and an OR. -- the reader should calculate for himself the results printed. Note that print instructions included in different DO;...;COMPUTE; blocks result in output separated by a blank line.

<u>Program 12</u>

```
LINE STATE
 NO   NO

          /* SOME MORE SET OPERATIONS   */
          DO;
          A=≤:1,2≥;
          B=≤:1,2,3,4≥;
          C=≤:≤:1,2≥,3,4≥;
          PRINT. B INCS, A;
          PRINT. C INCS, A;
          PRINT.A→C;
          PRINT.1→C;
          COMPUTE;
          DO;
          PRINT. A EQ, B;
          PRINT. B NE, C;
          PRINT.(1→A) AND, (4→B);
          PRINT.(23→A) OR, (4→C);
          PRINT.(4→B) AND, ((84→B) OR, (2→C));
          COMPUTE; FINISH;
```

<u>Output -- Program 12</u>

```
TRUE
FALSE
TRUE
FALSE

FALSE
TRUE
TRUE
TRUE
FALSE

* * * (END OF FILE ON INPUT) * * *
```

There is actually a primitive SETLB function which will form all of the subsets of a given set S. This always produces 2^n subsets. The list of subsets always includes the set itself and the null set. The operator which does this is the power function POW. The manner in which it is invoked is illustrated below. Note that more than one line is required for the entire printout.

Program 13

```
LINE STATE
 NO    NO

        /* THE SET OF SUBSETS */
        DO; A=≤:1,2,3,4≥;
        PRINT,POW(A);
        COMPUTE; FINISH;
```

Output -- Program 13

```
≤≤1≥ ≤2 3≥ ≤1 2 3≥ NL, ≤1 2≥ ≤3≥ ≤1 3≥ ≤2≥ ≤1 4≥ ≤2 3 4≥ ≤1 2
 3 4≥ ≤4≥ ≤1 2 4≥ ≤3 4≥ ≤1 3 4≥ ≤2 4≥≥

* * * (END OF FILE ON INPUT) * * *
```

Given a set S it is also possible by using a function called NPOW to prunt out only those subsets of S which contain a specified number of elements. In the program which follows the set A is defined and only those of its subsets which contain 3 elements are printed.

When using NPOW note that the first argument within the parentheses is the vlaue of N, followed by either the name of the set, or the set itself.

Program 14

```
LINE STATE
 NO    NO

        /* THE SET OF ALL SUBSETS HAVING N ELEMENTS */
        DO;
        A=≤:1,2,3,4,5≥;
        PRINT,A;
        COMPUTE; DC;
        PRINT,NPOW(3,A);
        COMPUTE; FINISH;
```

Output -- Program 14

```
≤1 2 3 4 5≥

≤≤1 4 5≥ ≤1 2 3≥ ≤2 4 5≥ ≤3 4 5≥ ≤2 3 4≥ ≤2 3 5≥ ≤1 3 5≥ ≤1 2
  4≥ ≤1 3 4≥ ≤1 2 5≥≥

* * * (END OF FILE ON INPUT) * * *
```

2.6 Union of Sets

One of the fundamental operations of set theory is that of union. This is represented in SETLB by the operator sign +. If

A = ≤:1,2≥; and B = ≤:3,4≥;

then the union A+B of A and B is

≤:1,2,3,4≥;

Difference of Sets

Next, consider

C = ≤:1,2,3≥; and D = ≤:1,2,4≥;

The difference of these two sets is the set which is left when all those elements of the set D which belong to set C are removed from C. Therefore,

(≤: 1,2,3≥ - ≤: 1,2,4≥) EQ. ≤:3≥

has the value TRUE.

Intersection of Sets

The collection of elements which two sets have in common with each other is known as the intersection of the two sets. This construction is represented in SETLB using the operator *, the asterisk. For example, if

C = ≤:1,2,3≥; and D = ≤:3,4,5≥;

then the intersection C*D of C with D will be:

≤:3≥

The above examples show how the operators +, - and *, which ordinarily are used for arithmetic purposes, are here used for the set operations of union, difference and intersection, respectively.

2.7 Symmetric Difference

In dealing with two sets we may wish to refer to those elements which are present in either one or the other but not both. This is the so-called 'symmetric difference' and is represented in SETL by the operator //, two adjacent slashes. If:

$$E = \underline{<}:1,2,4\underline{>}; \quad \text{and} \quad F = \underline{<}2, 3,5\underline{>};$$

then the symmetric difference of E and F will be:

$$\underline{<}:1,3,4,5\underline{>}$$

All of the above remarks are illustrated in the next program.

Program 15

```
LINE STATE
 NO    NO

          /* UNION, INTERSECTION, ETC. */
          DO;
          PRINT,(≤:5,6≥ + ≤:18,19≥) EQ, ≤:5,6,18,19≥;
          PRINT,(≤:1,2,3≥ - ≤:1,2,3,4≥) EQ, ≤:4≥;
          PRINT,≤:4≥ EQ, (≤:1,2,3,4≥ - ≤:1,2,3≥);
          PRINT,(≤:1,2,3,4≥ * ≤:2,4,6,8≥) EQ, ≤:2,4≥;
          PRINT,(≤:5,6,7,8≥ // ≤:4,6,8≥) EQ, ≤:4,5,7≥;
          COMPUTE; FINISH;
```

Output -- Program 15

```
TRUE
FALSE
TRUE
TRUE
TRUE

* * * (END OF FILE ON INPUT) * * *
```

2.8 The Number of Elements in a Set

In order to determine how many elements a set contains one can use the ↓ operator -- the downwards pointing arrow. (This is used instead of the sign #, which is not available on the 029 keypunch.) In the first block of the following program a set A of eight elements is defined; the set B has one element, while both sets C and D are defined to be the null set, two ways of writing the null set being shown.

Program 16

```
/*  TO FIND THE NUMBER OF ELEMENTS IN A SET */
DO;
A=≤:1,2,3,4,5,6,7,8≥;
B=≤:6≥;
C=≤:≥;
D=NL,;
PRINT,(↓A);
PRINT,(↓B);
PRINT,(↓C);
PRINT,(↓D);
COMPUTE;
DO;
PRINT,((↓A) EQ, 3);
COMPUTE;FINISH;
```

Output -- Program 16

```
8
1
0
0

FALSE

* * * (END OF FILE ON INPUT) * * *
```

2.9 Duplicate Elements

If a set A containing certain elements is added to another set having a number of elements in common with A, the duplicate elements are ignored. For example, let:

A = ≤:1,2,3,4≥;

B = ≤:1,3,5≥;

If we now calculate:

C = A + B;

we know from what has been said before that the result will not be:

C = ≤:1,1,2,3,3,4,5≥;

since duplicate elements are ignored. Instead, the result is:

C = ≤:1,2,3,4,5≥;

Note that C is a set of five elements. The actual number of elements present in C can, of course, be tested with the ↓ operator. This is done in the following program.

Program 17

```
LINE STATE
 NO    NO

        /* DUPLICATE ELEMENTS IN SETS ARE IGNORED */
        DO; A=≤:1,2,3,4≥; B=≤:1,3,5≥;C=A+B;PRINT,C;
        PRINT,((↓A)EQ, 4 AND,(↓B) EQ, 3));
        PRINT,(↓C) EQ,5;
        COMPUTE; FINISH;
```

Output -- Program 17

```
≤1 2 3 4 5≥
TRUE
TRUE

* * * (END OF FILE ON INPUT) * * *
```

2.10 Changing the Contents of Sets

Suppose that we have two sets:

$$A = \underline{<}:1,2,3\underline{>}$$

$$B = <:2\underline{>} \quad .$$

We can subtract set B from set A by means of the minus, -, operator. Writing C = A - B would produce

$$C = \underline{<}:1,3\underline{>} \quad .$$

As already explained we can add a set to another set by means of the plus, +, operator. Let

$$D = \underline{<}:4\underline{>}$$

Writing E = A + D would produce

$$E = \underline{<}:1,2,3,4\underline{>}$$

Suppose we now wish to delete an element from a set. For this purpose we can use the LESS. operator. To insert an element into a set the WITH. operator may be used.

All these operations are performed in the next program.

Program 18

```
LINE STATE
 NO    NO

/* ADDING ELEMENTS AND DELETING ELEMENTS */
DO;
A=≤:1,2,3≥;
B=≤:2≥;
C=A-B;
D=≤:4≥;
PRINT,C;
PRINT,A+D;
PRINT,(≤:1,2,3≥ - ≤:2≥));
PRINT,(≤:1,2,3≥ LESS,2));
PRINT,(≤:1,2,3≥ - ≤:2≥) EQ,≤:1,3≥;
PRINT,(≤:1,2,3≥ + ≤:4≥));
PRINT,(≤:1,2,3≥ + ≤:4≥) EQ,≤:1,2,3,4≥;
COMPUTE;
DO;
E=≤:4,5,6≥;
H=E LESS,5;
I=E LESS,7;
J=E WITH,7;
PRINT,H; PRINT, I; PRINT,J;
COMPUTE; FINISH;
```

Output -- Program 18

```
≤1 3≥
≤1 2 3 4≥
≤1 3≥
≤1 3≥
TRUE
≤1 2 3 4≥
TRUE

≤4 6≥
≤4 5 6≥
≤4 5 6 7≥

* * * (END OF FILE ON INPUT) * * *
```

The LESS. and WITH. operators can be combined in complex expressions. The following program shows just this.

Program 19

```
LINE STATE
 NO   NO

      /* COMPOUND INSERTIONS AND DELETIONS */
      DO;
      A= ((≤:1,2,3,4,5≥ LESS,4) WITH,6) EQ,≤:6,5,3,2,1≥;
      PRINT.A;
      B=(((≤:10,20,30≥ WITH.40)LESS.20)WITH.50)EQ.(≤:10,30,100,40,50≥
      LESS,100);
      PRINT.B;
      COMPUTE;FINISH;
```

Output -- Program 19

```
TRUE
TRUE

* * * (END OF FILE ON INPUT) * * *
```

2.11 The "Selection" or "Arbitrary Element" Operator

It is sometimes desirable to operate on elements of a set, but in an order that does not matter. For this purpose SETLB provides the ARB. operator, which selects "any old" element of a set. This does not select a truly random element, but rather the first one as stored internally within the computer. (The internal representation is seen when the set is printed out by a PRINT. statement.) Consequently, successive ARB. calls to the same set will always select the same element, namely the first.

Referring to the following program, a set A is defined and is immediately printed out. ARB. A is then invoked twice in succession and on each occasion the same element is selected. When the set A is tested for the presence of an arbitrary element of A, of course, the answer will be TRUE. Next the set A is changed by removing the arbitrary element of A from it. The ARB. operator is then applied to the new set and, as is seen from the printed output, the first element of the new set A is selected.

The rest of the program should be self-explanatory.

Program 20

```
LINE STATE
 NO    NO

          /* ARBITRARY ELEMENTS OF A SET */
          DO;
          A=≤1,5,9,16≥;
          PRINT. A;
          PRINT. ARB,A;
          PRINT. ARB,A;
          PRINT.(ARB,A)→A;
          PRINT.A LESS.(ARB,A);
          PRINT. ARB,(A LESS. (ARB,A));
          COMPUTE;
          DO;
          X=ARB.A;
          PRINT.(X EQ. 5) OR.(X EQ. 16) OR.(X EQ. 9) OR.(X EQ. 1);
          COMPUTE;
          DO;
          PRINT.NOT.(X→NL,);
          COMPUTE;FINISH;
```

Output -- Program 20

```
≤16 9 1 5≥
16
16
TRUE
≤9 1 5≥
9

TRUE

TRUE

* * * (END OF FILE ON INPUT) * * *
```

QUESTIONS

Chapter 2

1. Which of the following sets is invalid in SETLB:

   (a) ≤ : 4, 5, -3, 9≥

   (b) < : 3, -2, 7≥

   (c) ≤ : 7, 8, 4+2, 3,

   (d) ≤1, -1, 48-4≥

   (e) ≤ 1, -1, 4, -4 ≥

   (f) ≤ 5, 9, -2 ≥ ≥

   (g) ≤ : 9, 6, 2, ≤ 8, 5 ≥

   (h) ≤ : 3//3, <1, 7, 5, ≥, 3, 4, 1 2 ≥

   (i) ≤:'JACK', 'HENRY', 'MAX', 'STEVE', 'KATE'≥

2. Assuming

   S = ≤ : 1, 7, 4, 5, 'HI', 7/2 ≥

   which of the following operations yields a TRUE answer?

   (a) 1 → S

   (b) 3 → S

   (c) 2 → S

   (d) NOT.(4 → S)

   (e) NOT.((7//2) → S)

   (f) 'H' → S

   (g) ':' → S

   (h) NL. → S

   (i) ≤ : 7 ≥ → S

3. Assume

   S = ≤ : 4, 6, -1, 'JOE' ≥

   T = ≤ : 'JOE', -1 ≥

   which of the following Boolean expressions yields a TRUE answer?

   (a) S INCS. T

   (b) S INCS. ≤: 6 ≥

   (c) S INCS. ≤ : 1, 3, 5 ≥

(d) S INCS. NL.

(e) ≤ : 4, 6 ≥ INCS. S

(f) T INCS. ≤ : 'JOE' ≥

(g) T INCS. T

(h) S INCS. ≤ : 4, 6, 13/2, (-2*(3//2))/2, -1, -1,('JOE') ≥

4. Which of the following SETLB Boolean expressions yields a FALSE answer?

(a) 7 EQ. 6

(b) NOT.(4 NE. 4)

(c) 3 NE.(7//2)

(d) 4 GE.(12/3+1)

(e) (≤:4,5≥ INCS. NL.) OR. (5 LT. 8)

(f) N.((8/2) LE. (5+7/2))

(g) (8/2) GE. ((7//2) + 7/2)

(h) (T.O.((4+3) GT. (6-(-2))) A. (≤:F.,4≥ INCS. ≤:(4 LT. 2)≥))

5. Which of the following Boolean expressions are TRUE?

(a) (≤:1≥ + ≤:2≥) EQ. ≤:1,2≥

(b) (≤:1,2,3≥ - ≤:1,2≥) NE. ≤:3≥

(c) (≤:1,2,3≥ * ≤:3,4,5≥) EQ. ≤:3≥

(d) (≤:1,2,4≥ // ≤:2,3,5≥) NE. ≤:1,3,4≥

6. What is the value of the following expressions?

(a) ↓(≤:1,2≥ + ≤:3,4≥)

(b) ↓(≤:1,1,2,3≥ - ≤:1,2,3,4,5≥)

(c) ↓(≤:1,2,3≥ + ≤:3,4,5≥)

(d) ↓(≤:3,5,≤:1,2≥≥) + ↓(≤:1,2≥) + ↓(≤:1,2≥ + ≤:3,5≥)

(e) ↓NPOW(0, ≤:1,5,≤:1,2,7≥,'HI', 'HO'≥)

7. Write out the result of each of the following operations:

(a) ≤:1,2,3,4,5,6,7,8,9,100≥ LESS. (50*2)

(b) ≤:1,2,≤:3,4≥,5,6> WITH. (≤:6/2, 1+7/2≥)

(c) ≤:1,5,7,8≥ - ≤:1,4,7≥

(d) ≤:1,7,112 3≥ + ≤:1,12,3≥

8. Assume the set

S = ≤:1,5,9,16≥

is stored internally as

≤16 9 1 5>

Write the output you would expect from running the following SETLB program:

```
DO;
     S = ≤:1,5,9,16≥;
     PRINT. S;
     X = ARB. S;
     Y = S LESS. X;
PRINT. X,Y,(Y WITH. X);
     Z = ARB. Y;
PRINT. Z,Y LESS. Z, ARB. (Y LESS. Z);
COMPUTE; FINISH;
```

9. What results are obtained from the following Boolean expressions?

(a) ≤:1,2≥ → NPOW(2,≤:1,2,3,4≥)

(b) NL. → NPOW(1,≤:1,4,5,7≥)

(c) ≤:1,2≥ → POW(≤:1,4,5,7,3≥)

(d) ≤:1,1+2//1,-(2-3/2-1)≥ → POW(≤:1,4,5,7,3≥)

3. Tuples

SETLB provides not only sets but also tuples as basic data objects. Unlike sets, tuples are well defined, ordered sequences of components. For example,

<13,4,8>

is a tuple of three components. Arbitrary SETLB objects including sets or even other tuples can be components of a tuple. Tuples differ from sets in that the order in which the components appear is important.

One will often want to affix new components to tuples -- a process called concatenation. The tuple concatenation operator is designated by the familiar + (plus sign). One will often want to know how many components a particular tuple contains. The enumerator is the no less familiar ↓ (downward pointing arrow, used instead of the sign #, which is not available in the 029 keypunch).

Here is an elementary program involving tuples.

Program 21

```
LINE STATE
 NO    NO

           /*  OPERATIONS ON TUPLES */
           DO;A=<1,5,9>; B=<9,5,1>;
           PRINT.A EQ, B;
           COMPUTE;
           DO;C=<4,4,4>; D=<4,4>;
           PRINT. C NE, D;
           COMPUTE;
           DO; E=<≠HI≠,63,≤1,2,3≥>;
           PRINT.≠A VALID EXAMPLE OF A 3-TUPLE≠,E;
           COMPUTE;FINISH;
```

Output -- Program 21

```
FALSE

TRUE

≠A VALID EXAMPLE OF A 3-TUPLE≠ <≠HI≠ 63 ≤1 2 3≥>

* * * (END OF FILE ON INPUT) * * *
```

The symbols < > are used to delimit tuples; note the difference between these 'tuple brackets' and the symbols $\underline{<}$: $\underline{>}$, which are used for sets.

The first line of output from the preceding program makes it clear that the <u>tuples</u> A and B are not identical even though the <u>sets</u> of their components are identical. Nor for that matter are C and D equal. The tuple E is a valid 3-tuple even though its first component is a character string (enclosed by quote signs), its second is a numerical value, and its third component is a set.

When one tuple is concatenated with another the new tuple formed has a total number of components equal to the sum of the lengths of the original tuples, regardless of whether identical components are present or not. This is shown by the next program.

Program 22

```
LINE STATE
 NO    NO

      /* MORE ON TUPLES */
      DO; A=<1,2,3>; B=<1,2,3,4,5>;
      PRINT.((↓A)+(↓B)) EQ.(↓(A+B));
      C=A+B;
      PRINT.C;COMPUTE;FINISH;
```

Output -- Program 22

```
TRUE
<1 2 3 1 2 3 4 5>

* * * (END OF FILE ON INPUT) * * *
```

3.1 Indexing of Tuples

Since tuples are well defined, ordered sequences of components we can retrieve a tuple's components using numerical indices. The first component will be addressed by the index 1, the second 2, etc. To retrieve a desired component, its index is merely enclosed within parentheses following the name of the particular tuple. If either by error or design, the index is greater than the number of components of the tuple, OM. results. All this is shown in the next program, where it will be noticed that the last example involves a tuple followed by an index. In all such cases the tuple must be enclosed by parentheses to avoid syntactic problems.

Program 23

```
LINE  STATE
 NO    NO

            /* INDEXING OF TUPLES */
            DO;
            A=<2,4,8,16>;
            PRINT.(A(1) EQ. 2) AND. (A(4) EQ. 16);
            PRINT.(A(2) EQ. 7) OR. (A(2) EQ. 4);
            PRINT.A(5);
            PRINT.A(5) EQ. OM.;
            PRINT.(<7,4,6,2>)(5);
            COMPUTE; FINISH;
```

Output -- Program 23

```
TRUE
TRUE
OM.
TRUE
OM.

* * * (END OF FILE ON INPUT) * * *
```

3.2 The Zeroth Component of a Tuple: An Illustration of Error Termination

An attempt to retrieve the 0th, -1st, etc. component of a tuple causes an error termination or a "crash". (For a fuller explanation see Chapter 3.4.) This is deliberately demonstrated in the first block of the following program. Note that, on a crash, information intended to assist in debugging is printed out. Specifically the sequence of SETLB system-routine calls leading to a fatally offending operation is printed out. Since the crash shown in the following program occurred in the first block, the program was not aborted but continued to the next block in which A(5) was sought, alas in vain; thus an omega was returned.

In the third and last block B is defined to be a tuple containing components which are themselves tuples. B = <<4,6>,<1,3>,<5,8>>; . The third component could be printed out simply by writing:

```
PRINT. B(3);  .
```

This would give the tuple:

```
<5,8>  .
```

Suppose, however, that we want to see not this tuple but its first component. To do so all one has to write is:

```
PRINT. (B(3))(1);
```

this can be seen from the next program.

<u>Program 24</u>

```
LINE STATE
 NO   NO

/* MORE ON INDEXING OF TUPLES */
DO; A=<11,2,14,8>;
PRINT,A(3);PRINT,A(4);PRINT,A(0);
COMPUTE;
DO; PRINT,A(5);
COMPUTE;
DO; B=<<4,6>,<1,3>,<5,8>>;
PRINT,(B(3))(1);
COMPUTE;FINISH;
```

<u>Output -- Program 24</u>

```
14
8
≠RUN-TIME VALUE ERROR IN ≠ ≠TUPLE. OF. X, X LESS. THAN 1≠
≠*** CRASH ***≠
PROCEDURE TRACE - IN REVERSE CALLING SEQUENCE
OFTUPL.
        ARG 1 = <11 2 14 8>
        ARG 2 = 0
OF.
        ARG 1 = <11 2 14 8>
        ARG 2 = 0
OFN.
        ARG 1 = <11 2 14 8>
        ARG 2 = (0)
MAIN

OM.

5


* * * (END OF FILE ON INPUT) * * *
```

3.3 Modifying Tuples

To add new components to the end of a tuple one can simply use the concatenation operator. If, for example,

A = <1,2,3>;   and   B = <4>;

then A + B is <1,2,3,4>.

SETL provides another method allowing single components to be added to the tail end of a tuple. Suppose, for example, that

B = <10,20,30>;

and that we desire to attach the component 40 to the tail end of B. We can get to the last component by writing

B(↓B);

To add the component 40 after the present tuple end we can refer to the 'element after the last' and write

B((↓B) + 1) = 40;

In the next program the tuple A is concatenated to B. The printout clearly shows this to have been accomplished correctly. Note that concatenating <8> to <5,6,7>, in this order, does not produce the tuple <5,6,7,8> but rather <8,5,6,7>, something quite different.

In the second block the fourth component of tuple B is set equal to 40, as described above. A printout of the new B shows the result. Finally, in the third block, the last component of the new B is modified.

Program 25

```
LINE STATE
 NO    NO

          /* ADDING A COMPONENT TO THE END OF A TUPLE */
           DO; A=<1,2,3>;  B=<4>;
           PRINT.A+B;
           PRINT.(<8> + <5,6,7>) EQ, <5,6,7,8>;
           COMPUTE;
           DO; B=<10,20,30>; PRINT.B;
           B((+B)+1)=40;
           PRINT.≠NOW B IS ≠,B;
           COMPUTE; DO;
           B(+B)=50;
           PRINT.≠NOW B IS≠,B;
           COMPUTE;FINISH;
```

Output -- Program 25

```
<1 2 3 4>
FALSR

<10 20 30>
≠NOW B IS ≠ <10 20 30 40>

≠NOW B IS≠ <10 20 30 50>

* * * (END OF FILE ON INPUT) * * *
```

As is shown in the preceding example, the components of a tuple can be modified directly by indexing, in very much the same way as arrays are modified in conventional programming languages. The next program illustrates this.

<u>Program 26</u>

```
LINE STATE
 NO    NO

           /* MODIFYING A TUPLE */
           DO;
           A=<1,3,5,7,9>;
           PRINT.A;
           A(1)=100;
           PRINT.A;
           A(2)=80;A(4)=78;
           PRINT.A;
           COMPUTE; FINISH;
```

<u>Output -- Program 26</u>

```
<1 3 5 7 9>
<100 3 5 7 9>
<100 80 5 78 9>

* * * (END OF FILE ON INPUT) * * *
```

The next program gives another illustration of the modification of a tuple. It should be self-explanatory.

Program 27

```
LINE STATE
 NO   NO

          /* MODIFYING A TUPLE */
          DO;
          A=<1,3,5,7,9>;
          PRINT,A;
          A(1)=10;
          PRINT,A;
          A(2)=20; PRINT,A;
          PRINT,A(↓A);
          PRINT,A((↓A)+1);
          A(↓A)=100; PRINT,A;
          A((↓A)+1)=200; PRINT,A;
          A((↓A)+2)=300; PRINT,A;
          B=A; PRINT,B;PRINT,B(7);
          COMPUTE; FINISH;
```

Output -- Program 27

```
<1 3 5 7 9>
<10 3 5 7 9>
<10 20 5 7 9>
9
OM
<10 20 5 7 100>
<10 20 5 7 100 200>
<10 20 5 7 100 200 OM 300>
<10 20 5 7 100 200 OM 300>
OM

* * * (END OF FILE ON INPUT) * * *
```

3.4 The 'Undefined Value' or 'Omega' Concept; Additional Remarks on Error Termination.

Whenever an operation is attempted which violates the formal rules of SETLB, it will cause either a 'crash', i.e., an error termination, or it will produce an 'indefinite', an 'omega', represented in SETLB by the symbol OM. . In general, 'plausible' violations will produce OM. ; 'implausible' violations will lead at once to an error termination. Note that the presence of an omega can be exploited in a program; for example, an element may be tested for equality with omega.

In the following programs, A and B are valid tuples of length 2. If we try to retrieve their third components A(3), B(3), we get the 'undefined' result twice. (Note that, as shown earlier, an attempt to retrieve the zeroth, or a negative, component of a tuple does not retrieve an omega but causes a crash.)

Generally speaking, the use of an operator with invalid arguments, as e.g., in the combination

```
7 INCS. 6
```

will also lead to a crash.

All this is shown in the next program, which also shows how the system recovers from crashes.

Program 28

```
LINE STATE
 NO   NO

          /* OMEGA AND ERROR TERMINATICNS */
          DO; A=<2,3>; B=<5,6>;
          PRINT,A(3),B(3),B(5),A(2),B(1);
          COMPUTE; DO;
          PRINT,≠NOW AN IMPLAUSIBLE OPERATION≠;
          PRINT, 5 INCS, 6;
          PRINT,≠THIS MESSAGE WILL NEVER APPEAR≠;
          COMPUTE; DO;
          PRINT,≠BACK IN BUSINESS AFTER A CRASH≠;
          PRINT,≠NOW FOR ANOTHER CRASH≠;
          PRINT,A*B;
          COMPUTE; DO;
          PRINT,≠RECOVERY IS POSSIBLE AFTER ANY NUMBER OF CRASHES≠;
          COMPUTE; FINISH;
```

Output -- Program 28

```
OM, OM, OM, 3 5

≠NOW AN IMPLAUSIBLE OPERATION≠
≠INVALID DATA TYPE, FOR THE SETL OPERATION: ≠ ≠X INCS, Y, Y OR X NOT A
SET,≠
≠*** CRASH ***≠
PROCEDURE TRACE - IN REVERSE CALLING SEQUENCE
INCS,
      ARG 1 = 5
      ARG 2 = 6
MAIN

≠BACK IN BUSINESS AFTER A CRASH≠
≠NOW FOR ANOTHER CRASH≠
≠INVALID DATA TYPE, FOR THE SETL OPERATION: ≠ ≠A*B≠
≠*** CRASH ***≠
PROCEDURE TRACE - IN REVERSE CALLING SEQUENCE
TMS,
      ARG 1 = <2 3>
      ARG 2 = <5 6>
MAIN

≠RECOVERY IS POSSIBLE AFTER ANY NUMBER OF CRASHES≠

* * * (END OF FILE ON INPUT) * * *
```

3.5 The Head and Tail Operators

The head of a tuple is its first component. The tail is the tuple which remains after the head is removed. These portions of a tuple may be produced directly by means of the SETLB HD. and TL. operators.

This is shown in the following example. We let

TUP = <5,6,8,4>

Printing HD. TUP gives the number 5, The instruction:

PRINT. TL. TUP;

prints out the tuple

<6 8 4>

Since the tuple TUP has four components, the tail of the tail of the tail of the tail of TUP is the null tuple, designated as NULT. Anything beyond this gives an omega, as is shown by the following program.

Program 29

```
LINE STATE
 NO   NO

        /* THE HEAD AND TAIL OPERATORS */
        DO;
           TUP=<5,6,8,4>;
        PRINT.TUP;
        PRINT.HD.TUP;
        PRINT.TL.TUP;
        PRINT.HD.TL.TUP;
        PRINT.HD.TL.TL.TUP;
        PRINT.TL.TL.TL.TUP;
        PRINT.TL.TL.TL.TL.TUP;
        PRINT.HD.TL.TL.TL.TL.TUP;
        PRINT.TL.TL.TL.TL.TL.TUP;
        COMPUTE; FINISH;
```

Output -- Program 29

```
<5 6 8 4>
5
<6 8 4>
6
8
<4>
NULT.
OM.
OM.


* * * (END OF FILE ON INPUT) * * *
```

QUESTIONS

Chapter 3

1. Which of the following Boolean expressions are TRUE?

   (a) <1,1,4,5,7> EQ. <1,5,4,7>

   (b) <4,7,9,1,2> NE. <1,7,4,7,9>

   (c) (((<1,2,3,5,6,7,4>)(3+3) NE. (3+3))

   (d) ((((<1,2,3,<1,2,3,4> 7,8>)(4))(3)) EQ. (7/2))

   (e) (((<1,4,5,8,<1,4,7>3,5,6>)((<1,7,3,2>)(2))) EQ. (5))

   (f) (<3,5,7,8,9> + <7,8,9,3,5>) EQ. <3,5,7,8,9>

   (g) (<1,2,3,4,5> + NULT.) EQ. <1,2,3,4,5>

   (h) (<1,2,3,4,5> + <3,5,6,7>) NE. <1,2,3,4,5,3,5,6,7>

   (i) (((<1,2,3,5,6,>)(6)) NE. (OM.))

   (j) (((<1,3,7,9,5,3>)((<3,5,7,0,8,9,10,3,7,2>)(7/2)))
       EQ. ((<3,5,6,OM.>)(4)))

4. ADDITIONAL INFORMATION ON SETS AND TUPLES

4.1. Set Formers

In defining sets we have till now specified them explicitly. For example, we have written

```
A = ≤:1,2,3,4≥;
PRINT. A;
```

Given a set (and no matter how it was originally formed) we will often want to form certain of its subsets. For example, we might be interested in knowing all of the elements X of A which are greater than 2. This can be done very directly using the following expression:

```
PRINT. ≤X → A ↑ X GT. 2≥;
```

where the upwards pointing arrow stands for "such that." The result is obviously ≤:3,4≥. Note that when a set is defined by such a set former the colon is omitted.

Similarly if set B is defined as:

```
≤:10,20,30,40,50≥;
```

we might want to know all the elements Y of set B such that Y is less than or equal to 30. This set may be built quite similarly:

```
PRINT. ≤Y → B ↑ Y LE. 30≥;
```

The elements which satisfy the stated condition form the set ≤:10,20,30≥.

Both the above examples are shown in the next program, which also contains a third set former. As the omega printed by the last line shows, the use of a variable in a set former does not affect the value of the variable in any predictable way.

Program 30

```
LINE STATE
 NO   NO

/* SOME ELEMENTARY SET FORMERS */
DO;  A=≤:1,2,3,4≥;  PRINT,A;
PRINT,≤X→A↑X GT, 2≥;
COMPUTE;
DO;  B=≤:10,20,30,40,50≥; PRINT,B;
PRINT,≤Y→B↑Y LE,30≥;  COMPUTE;
DO;  C=≤:-3,-2,-1,0,1,2,3≥;  PRINT,C;
PRINT,≤Z→C↑Z LT,0≥;
PRINT,Z;D=≤ Z→C↑Z LT,0≥;PRINT,D;COMPUTE;FINISH;
```

Output -- Program 30

```
≤1 2 3 4≥
≤3 4≥

≤40 50 10 20 30≥
≤10 20 30≥

≤0 1 -1 2 -2 3 -3≥
≤-1 -2 -3≥
OM.
≤-1 -2 -3≥

* * * (END OF FILE ON INPUT) * * *
```

Consider next a set A defined explicitly as follows:

≤:2,3,4,5,6,7,8,9≥

This is the set of integers greater than one but less than ten. This set may be defined directly using a set former, without any explicit enumeration being required. We need merely write:

≤N, 1 < N < 10≥ .

The same set can be constructed by using a differently worded statement. Let C be the set of elements M, where M is greater than or equal to 2 and less than or equal to 9. This set, which we may display by writing

PRINT. ≤M, 2 <= M <= 9≥;

is clearly the same as A.

Note that in SETLB the double operators <= and <u>not</u> the ≤ sign, which is used strictly to delimit sets, must be used to write "less than or equal".

Naturally, all of the sets mentioned above are equal; this is confirmed by the printed output of the next program.

Program 31

```
LINE STATE
 NO    NO

/* INTEGER RANGES IN THE SET FORMER */
DO;
A=≤:2,3,4,5,6,7,8,9≥;
PRINT,A;
B=≤N,1<N<10≥;
PRINT,B;
C=≤M,2<=M<=9≥;
PRINT,C;
PRINT,(A EQ, B) AND, (B EQ,C);
COMPUTE; FINISH;
```

Output -- Program 31

```
≤8 9 2 3 4 5 6 7>
≤8 9 2 3 4 5 6 7≥
≤8 9 2 3 4 5 6 7≥
TRUE

* * * (END OF FILE ON INPUT) * * *
```

4.2. A Short Digression on Arithmetic in SETLB

A useful remainder operator is provided in SETLB. The operator is written using two adjacent slashes, i.e. the remainder obtained on dividing A by B is written in SETLB as:

A//B

This remainder operator is illustrated in the next program.

Program 32

```
LINE STATE
 NO   NO

        $
        /* THIS IS TO ILLUSTRATE THE REMAINDER OPERATOR */
        $
        DO;
        A=9; B=2;
        PRINT,≠THE REMAINDER OF A/B=≠,A//B;
        COMPUTE; FINISH;
```

Output -- Program 32

```
≠THE REMAINDER OF A/B=≠ 1

* * * (END OF FILE ON INPUT) * * *
```

In the current implementation of SETLB, integer numbers only are permitted. This implies that built-in trigonometric functions, as well as many of the mathematical functions provided in languages such as Fortran, etc., are not available to the SETLB user. All arithmetic is done in the integer mode; the presence in SETLB source programs of a number with a decimal point could lead to unsuspected errors. The largest integer possible in the present implementation is $2^{18}-1$, which is 262,143.

SETLB provides no exponentiation operator. Of course, one can express exponentiation by repeated multiplication. Multiplication is designated by the sign *, the slash, /, is used for division. Of course, the plus, +, and minus sign, -, are used to designate addition and subtraction respectively.

Division by zero, even division of zero by zero yields a zero result, and is in no way flagged as an error.

It is hoped that by the end of 1973 a new version of SETLB, eliminating all the above difficulties and restrictions, will become available.

4.3. More Examples of Set-Formers.

We now return to continue the development of the set former concept. In the next program a set A containing the integers 1 through 10 is formed. This set is printed.

Next the set B of integers 1 through 10 such that the remainder of N modulo 2 is equal to zero is formed. These are none other than the even numbers 2, 4, 6, 8 and 10, which are then printed out.

After this we ask whether it is true or false that the set of all numbers N * N - 3 for N between six and eight inclusive is equal to the set {33, 46 and 61}. It clearly is; the printout confirms this. This is seen on the output line labelled 'the value for test 1'.

Next, a set C is defined and we ask whether it is true that for every X of C, 2*(X+3) is a member of the set {8,10,12}. Inspection shows this to be true; its truth is confirmed by 'test 2' of the following program.

Referring once again to the set C we ask whether for every X of C and for all N greater than 6 but less than 10, X + N is a member of the set {8,10,12}. Since the set of all such X + N is the set {8,9,10,11,12}, we get FALSE as the printout from 'test 3'.

Finally, we form the set of all N greater than 2 and less than or equal to 10 such that the remainder of N/3 is equal to 1. This set is saved under the name D and then printed out.

Caution is advised when expressions like N//2 and N//3 are used in a Boolean expression. In order to avoid difficulties the user is strongly advised to parenthesize all such operations.

Program 33

```
LINE STATE
 NO    NO

        /* MORE ON THE SET FORMER */
        DO;
        A=≤N,0<N<=10≥;
        PRINT,A,≠ WHICH IS THE SET A≠;
        B=≤N, 0< N <= 10 ↑(N//2)EQ, 0≥;
        PRINT,B,≠WHICH IS THE SET B≠;
        PRINT,≤(N*N)-3,6<=N<=8≥ EQ,≤:33,46,61≥,≠THE VALUE FOR TEST1≠;
        COMPUTE; DO;
        C=≤:1,2,3≥;
        PRINT,C,≠WHICH IS THE SET C≠;
        PRINT,≤2*(X+3),X→C≥ EQ, <:8,10,12≥,≠WHICH IS THE VALUE FOR TEST 2≠;
        PRINT,≤X+N,X→C,6<N<10≥ EQ,≤:8,10,12≥,≠WHICH IS THE VALUE FOR TEST 3≠;
        COMPUTE; DO;
        D=≤N,2<N<=10 ↑(N//3)EQ,1>;
        PRINT,D,≠ WHICH IS THE VALUE FOR SET D≠;
        COMPUTE; FINISH;
```

Output -- Program 33

```
≤8 9 1 10 2 3 4 5 6 7≥ ≠ WHICH IS THE SET A≠
≤8 10 2 4 6≥ ≠WHICH IS THE SET B≠
TRUE ≠THE VALUE FOR TEST1≠

≤1 2 3≥ ≠WHICH IS THE SET C≠
TRUE ≠WHICH IS THE VALUE FOR TEST 2≠
FALSE ≠WHICH IS THE VALUE FOR TEST 3≠

≤10 4 7≥ ≠ WHICH IS THE VALUE FOR SET D≠


* * * (END OF FILE ON INPUT) * * *
```

4.4. <u>Existentials</u>.

One will often wish to ask whether an element with some given particular property exists within a set. Consider, for example, the explicitly defined set:

$$A = \underline{<}:1,2,4,6\underline{>}$$

We can ask, for example, whether there exists an element X of the set A such that X is less than zero. For the above set A, this is obviously false. Next we can ask whether there exists an element X in A such that X is equal to 4. This is certainly true.

The symbol ≡ is used for testing existence; this replaces the conventional mathematical symbol ∃, which is not available on the 029 keypunch. Using this symbol, the two questions posed above may be written as follows in SETLB.

```
PRINT. ≡ X → A ↑ X LT. 0;
PRINT. ≡ X → A ↑ X EQ. 4;
```

Various existential expressions are shown in the following program.

Program 34

```
LINE STATE
 NO   NO

          /* EXISTENTIALS */
          DO;
          A=≤:1,2,4,6≥;
          B=≤:3,-5,7,-10≥;
          PRINT.≡ X → A ↑ X LT. 0;
          PRINT.≡ X → B ↑ X LT. 0;
          PRINT.≡ X → A ↑ X EQ. 4;
          PRINT.≡ X → A ↑ X EQ. 5;
          PRINT.≡ X → B ↑ X EQ. -10;
          PRINT. NOT.(≡ X → B ↑ X EQ. 100));
          COMPUTE; FINISH;
```

Output -- Program 34

```
FALSE
TRUE
TRUE
FALSE
TRUE
TRUE

* * * (END OF FILE ON INPUT) * * *
```

The existential quantifier has another important feature. When it is used to test whether an element with a given property exists in a set, it locates the first such element which it finds.

For example, let us define a set:

A = ≤:1,2,5,-8,9≥;

In the program which follows we ask whether there is an element X in A such that X is less than zero. The result is TRUE; the element -8 is less than zero and therefore after execution of the instruction X takes on the value -8.

This is shown in the next program.

<u>Program 35</u>

```
LINE STATE
 NO    NO

        /* TO ILLUSTRATE THAT THE EXISTENTIAL LOCATES */
        DO;
           A=≤:1,2,5,-8,9≥;
        PRINT.A;
        PRINT.∃X→A*X LT.0;
        PRINT.X;
        COMPUTE; FINISH;
```

<u>Output -- Program 35</u>

```
≤-8 9 1 2 5≥
TRUE
-8

* * * (END OF FILE ON INPUT) * * *
```

4.5. Universal Quantifier

Consider the set:

A = ≤:1,2,3,4≥

We may obviously assert that every element X of A is greater than zero. This assertion can be verified in SETLB by evaluating the following expression:

PRINT. ∨ X → A ↑ X GT. 0;

Note that the symbol for "for all" is the ∨, the 11-2-8 punch, not the letter V. (The symbol ∨ is used instead of the '∀' symbol conventional in mathematics because '∀' is not available on the 029 keypunch.)

If we wanted to know whether it is true that every element X of A is less than 2, we could similarly write:

PRINT. ∨ X → A ↑ X LT. 2;

These and other universal quantifiers are illustrated in the next program.

Program 36

```
LINE STATE
 NO    NO

        /* UNIVERSAL QUANTIFIERS */
        DO;
        A=≤:1,2,3,4≥;B=≤:2,-3,4,7≥;
        PRINT.∨ X → A ↑ X GT, 0;
        PRINT.∨ X → B ↑ X GT, 0;
        PRINT.∨ X → A ↑ X LT, 2;
        PRINT.∨ X → B ↑ X GE, 7;
        COMPUTE;FINISH;
```

Output -- Program 36

```
TRUE
FALSE
FALSE
FALSE

* * * (END OF FILE ON INPUT) * * *
```

The universal quantifier, like the existential quantifier, can be used for locating an element, though its use in this way is less natural. The element located by $\forall X \to A \uparrow C(X)$ is the first element x for which C(X) is <u>false</u>.

For example, let us again define the set A:

A = ≤:1,2,5,-8,9≥;

As will be seen from the next program, this set is stored as

≤-8 9 1 2 5≥

We then ask whether for all elements X of the set A, X is less than zero; the answer is FALSE. The first stored element, -8, certainly is less than zero, but the next element, 9, fails to satisfy the condition and so X takes on the value 9. This is seen in the next program.

<u>Program 37</u>

```
LINE STATE
 NO   NO

          /* TO ILLUSTRATE THAT THE UNIVERSAL
             QUANTIFIER ALSO LOCATES           */
          DO;
             A=≤:1,2,5,-8,9≥;
          PRINT,A;
          PRINT,∀X→A↑X LT,0;
          PRINT,X;
          COMPUTE; FINISH;
```

<u>Output -- Program 37</u>

```
≤-8 9 1 2 5≥
FALSE
9

* * * (END OF FILE ON INPUT) * * *
```

Let us now combine the two quantifier forms -- the existential and the universal quantifier. In the following example we use two sets:

$$A = \underline{<}:1,2,3,4\underline{>};$$

$$B = \underline{<}:3,4,5,6,7\underline{>};$$

Is it true that for every element X in A there exists an element Y in B such that X is equal to Y? We put this question in SETLB notation by writing:

```
PRINT. ∀ X → A ↑ (∃ Y → B ↑ X EQ. Y);
```

Then we ask: is it true that for every element X of A there exists an element Y of B such that X is not equal to Y? In SETLB this is:

```
PRINT. ∀ X → A ↑ (∃ Y → B ↑ X NE. Y);
```

These and a few other examples are included in the next program.

Program 38

```
LINE STATE
 NO   NO

/* MORE ON UNIVERSALS AND EXISTENTIALS */
DO;
A=≤:1,2,3,4≥; B=≤:3,4,5,6,7≥;
PRINT,≠A=≠,A;  PRINT,≠B=≠,B;
PRINT,∀X→A↑(∃Y→B↑X EQ,Y); PRINT,X;
PRINT,∀X→A↑(∃Y→B↑X NE,Y);
PRINT,NOT,(∃Y→A↑(∀X→B↑X EQ,Y));
PRINT,∀X→B↑X GT,0; PRINT,X;
PRINT,∃X→A↑X GT,0; PRINT,X;
COMPUTE; FINISH;
```

Output -- Program 38

```
≠A=≠ ≤1 2 3 4≥
≠B=≠ ≤3 4 5 6 7≥
FALSE
1
TRUE
TRUE
TRUE
OM,
TRUE
1


* * * (END OF FILE ON INPUT) * * *
```

Our final program illustrating the concept and use of universal and existential quantifiers is, for the sake of simplicity, broken up into eight parts.

(a) First we ask: is it true that for all integers N greater than or equal to 2 and less than or equal to 8, N is greater than zero? It certainly is, and we get a printout TRUE.

(b) Is it true that there exists an integer N greater than or equal to 5 and less than or equal to 8, such that N is equal to 6. Since N varies over all the integers 5 through 8 it certainly includes the integer 6, and once again we get the result TRUE. Printing out the value of N gives the result 6.

(c) Let

A = <:3,5,7,31≥;

and

B = <:1,-7,8,-44≥;

Is it true that for all N greater than zero and less than 5, N is a member of A? Since N varies over the integers 1 through 4 it is quite clear that not every value which N takes on is in A. Therefore the answer is FALSE. If after making this test we print out N, the number 1, i.e., the <u>first</u> value which <u>violates</u> the condition, is obtained.

(d) Is it true that for all N greater than 1 and less than 25, N is a member of A or is <u>not</u> a member of A? This statement is, of course, true, largely because of its second clause.

(e) Is it true that for every X which is an element of A, and for every element Y of B, X+Y is greater than -5? This may be found on inspection to be false. Note that in the program which follows only one "for all" symbol is used; SETLB allows a sequence of like quantifiers to be "run together" in the fashion indicated. Since the answer is FALSE, X is set to the first value which violates the condition -- in this case 3.

(f) Does there exist an element X of A such that X is equal to 123? The answer is FALSE. In this case the value of X is undefined and therefore an OM. is printed.

(g) Is it true that for every element X of A there exists an N between 1 and 4 such that X is equal to N? Since A contains 5, 7 and 31 the result must be FALSE. Printing out X yields the number 5, the first violation of the condition.

(h) Is it true that there exists an element X in A and there exists an element Y in B such that X is not equal to Y? The answer obtained is TRUE. Note the use of only one "there exists" symbol, allowed for the same reason that a single "for all" symbol is sufficient in connection with (e) above.

With these hints, the program which follows should be intelligible.

<u>Program 39</u>

```
LINE STATE
 NO    NO

/* MORE UNIVERSALS AND EXISTENTIALS */
DO;
PRINT.∀2<=N<=8↑N GT.0;
PRINT.∃5<=N<=8↑N EQ.6; PRINT.N;
      A=≤:3,5,7,31≥;
PRINT.∀0<N<5↑N→A; PRINT.N;
PRINT.∀1<N<25↑N→A OR.NOT.N→A;
      B=≤:1,-7,8,-44≥;
PRINT.∀X→A,Y→B↑(X+Y)GT.-5; PRINT.X;
PRINT.∃X→A↑X EQ.123; PRINT.X;
PRINT.∀X→A↑∃1<=N<=4↑X EQ.N; PRINT.X;
PRINT.∃X→A,Y→B↑X NE.Y; PRINT.X;
COMPUTE; FINISH;
```

<u>Output -- Program 39</u>

```
TRUE
TRUE
6
FALSE
1
TRUE
FALSE
3
FALSE
OM.
FALSE
5
TRUE
3


* * * (END OF FILE ON INPUT) * * *
```

4.6. 'Multiple' Assignments

Suppose that we write:

<X,Y,Z> = <5,10,15>;

This has the effect of assigning 5 to X, 10 to Y and <15> to Z as seen in the following program.

Program 40

```
LINE STATE
 NO   NO

      /* ASSIGNMENTS USING TUPLES */
      DO;<X,Y,Z>=<5,10,15>;
      PRINT,X,≠IS X≠;
      PRINT,Y,≠IS Y≠;
      PRINT,Z,≠IS Z≠;
      COMPUTE; FINISH;
```

Output -- Program 40

```
5 ≠IS X≠
10 ≠IS Y≠
<15> ≠IS Z≠

* * * (END OF FILE ON INPUT) * * *
```

What is the effect of writing:

<A,B> = <1,3,5,7>;

where the tuple on the left has two components and that on the right four? This assignment gives A the value 1 while the value of B becomes the tuple <3,5,7>.

Next, consider a set X defined to be:

X = ≤:1,3,6≥;

If we calculate the value X WITH. 9 we obtain a set to which the element 9 has been added. Note however that this calculation does not affect the value of X; the following program shows us clearly that the original set X remains intact after X WITH. 9 is calculated.

Of course, the assignment X = X WITH. 9 will change the value of the variable X. All this is illustrated by the next program.

Program 41

```
LINE STATE
 NO    NO

/* SOME MORE ASSIGNMENTS AND CALCULATIONS */
DO;
<A,B>=<1,3,5,7>;
PRINT.A,≠IS A≠;
PRINT.B,≠IS B≠;
COMPUTE;
DO;
X=$:1,3,6≥;
PRINT.(X WITH,9); PRINT.X;
X=X WITH,9; PRINT.X;
COMPUTE; FINISH;
```

Output -- Program 41

```
1 ≠IS A≠
<3 5 7> ≠IS B≠

$9 1 3 6≥
$1 3 6≥
$9 1 3 6≥

* * * (END OF FILE ON INPUT) * * *
```

Consider the tuple:

A = <10,20,30>;

To assign 5 to A(1), 'HULLO' to A(2) and 'GOODBYE' to A(3) is perfectly legal. Subsequent to this we may for example assign the tuple <4,5> to A(1). In each case a printout of A would show an appropriate component of A to have been modified.

<u>Program 42</u>

```
LINE STATE
 NO    NO

          /* INTERESTING EXAMPLES OF ASSIGNMENTS */
          DO;A=<10,20,30>;  PRINT,A;
          A(1)=5;A(2)=≠HULLO≠;A(3)=≠GOODBYE≠;
          PRINT,A;A(1)=<4,5>;PRINT,A;COMPUTE;FINISH;
```

<u>Output -- Program 42</u>

```
<10 20 30>
<5 ≠HULLO≠ ≠GOODBYE≠>
<<4 5> ≠HULLO≠ ≠GOODBYE≠>

* * * (END OF FILE ON INPUT) * * *
```

QUESTIONS

Chapter 4

1. Assume the set

S = ≤:1,5,-6,0,4,9≥

What set is generated by the following combinations?

(a) ≤X → S ↑ X LT. 0>

(b) ≤T → S ↑ T GT. 0>

(c) ≤U → S ↑ U EQ. 0>

(d) ≤V → S ↑ V NE. 0>

(e) ≤N, 1 < N < 10≥

(f) ≤M, 2 <= M <= 9≥

(g) ≤P, 3 < P < 8≥

(h) ≤Q, (3/2) <= Q <= (10/3)≥

2. How will SETLB, as described in this chapter, evaluate the following arithmetic expressions:

(a) 3 * 2 + 1

(b) 4 + 3 * 2

(c) 7/2

(d) 7//2

(e) 6 * 6 + 8 * 8

(f) (4*(19/2)) * 2

(g) (4*19/2) * 2

(h) 1 + 4 - 5 * 6/2 * 3

3. What set is generated by the following set formers:

(a) A = ≤B, 0 < B < 10≥

(b) B = ≤C, 1 <= C <= 10≥

(c) C = ≤D, 0 <= D <= 10 ↑ (D//2) EQ. 0≥

(d) D = ≤E, 2 < E <= 10 ↑ (E//3) EQ. 0≥

4. Assume the following sets:

```
S = ≤:1,3,-2,-9,11≥
T = ≤:1,3,5,-3,21,-6,-10≥
U = ≤N, -10 < N < 10≥
```

Which of the following Boolean expressions are TRUE?

```
(a)  ≡ N → U ↑ N LT. 0
(b)  ≡ N → T ↑ (N*N) LT. 0
(c)  ≡ N → U ↑ N EQ. (↓S)
(d)  ≡ N → S ↑ N → (T*U)
(e)  ≡ N → U ↑ T INCS. ≤: (N-3), (N-1)≥
(f)  NOT. (≡ N → T ↑ (N LT. 5) A. (N GE. 3))
(g)  ≡ Q → S ↑ ((Q EQ. -3) O. (Q) → (T*U)))
(h)  NOT. (≡ Q → U ↑ Q → ≤ N*N, 1 < N <= 10≥)
```

5. Assume the following sets:

```
S = ≤1,2,<3>,≤:4≥,<<5,6 >,<7,8>>,9,≤N*N, 1 <= N <= 10≥≥
T = ≤:3,5,8,9,12,-3,-5,-8,-9,-12,0≥
U = ≤:3,5,11,15,-7,30100,-5,71,-11,-12,-3,1≥
```

What will the following statements PRINT.?

```
(a)  PRINT. ∨ N → T ↑ N → (T * S)
(b)  PRINT. ≡ N → T ↑ ∨M → U ↑ N LT. M;
(c)  PRINT. ≡ M → U ↑ ∨N → T ↑ M GE. N, M;
(d)  PRINT. ≡ N → U, M → T ↑↑≤:N≥ INCS. ≤:M≥;
(e)  PRINT. ∨ N → U ↑ ≡ M → T ↑ <N,M> → S;
(f)  PRINT. ≡ N → S, 1 <= M <= 10 ↑ ≤:N≥ INCS. ≤:M≥;
(g)  PRINT. ∨ N → T, M → U ↑ N NE. M;
```

6. What will be printed by the following program?

```
DO;
A = 7*5-2;
B = 3+3;
C = 8;
S = <A,B,<C>>;
PRINT. A,B,C,S;
A = 7;
B = 8;
C = <C>;
PRINT. A,B,C,S;
A = 3;
B = 4;
C = 5;
<A,B,C> = S;
PRINT. A,B;
PRINT. 'C IS', C;
COMPUTE;
FINISH;
```

7. What will the following programs PRINT ?

a.
```
DO;
<A,B,C> = <<1,2>, <3,4>, 5,6>;
PRINT. A,B;
<A,B> = <3,4,5,6>;
PRINT. A,B;
PRINT. ≠C is ≠,C;
COMPUTE;
FINISH;
```

b.
```
DO; <X,Y,Z> = <5,10,15>;
PRINT. X, ≠IS THE VALUE OF X≠;
PRINT. Y, ≠IS THE VALUE OF Y≠;
PRINT. Z, ≠IS THE VALUE OF Z≠;
<X,Y,Z> = <7,<8,9,10,11>,12,13,14,15,≤:X,Y,Z≥>;
PRINT. <X,Y,Z>,X,Y,Z;
COMPUTE;
FINISH
```

c.
```
DO;
X = ≤:1,2,3,4≥
Y = X WITH. (5);
PRINT. ↓X, ↓<ARB. X>
X = Y;
PRINT. ↓X, ↓Y, ↓(X//Y);
COMPUTE;
FINISH;
```

8. Write an instruction which will print out the set of all primes which divide a given number N, and an instruction which will print the set of all primes whose squares divide a given number N.

9. Write a program which will calculate and print the set of all 'prime pairs' up to a given N. These are the pairs <P,Q> such that Q EQ. (P+2) and both P and Q are primes. Don't make your program unnecessarily inefficient!

5. SETS OF PAIRS AND TUPLES USED AS MAPS

5.1. Sets of Tuples as Functions

Mathematically speaking, a function of one variable can be identified with a set of ordered pairs. For example, the function which maps each integer n into its square n^2 can be identified with the following set of ordered pairs:

```
{<1,1>, <2,4>, <3,9>, ..., <-1,1>, <-2,4>, ...}
```

Of course, this mathematical convention (which many readers will have met in elementary mathematics courses) is more suited for theoretical than for practical purposes. In particular, functions like the function n^2, which are defined for all integers, will always correspond to *infinite* sets of ordered pairs; and since SETLB deals only with finite sets, it is not really possible in SETLB to regard every function as a set of ordered pairs. However, it is possible to regard every set of ordered pairs as defining a function; but such functions will always be defined only on a finite domain. Moreover, they can be multivalued. In the present section we will explain how SETLB uses sets of ordered pairs as functions.

First consider the set F defined by

```
≤:<'CAT',4>, <'MAN',2>, <'BIRD',2>, <'FLY',6>≥;
```

For each of the four creatures 'CAT', 'MAN', 'BIRD', and 'FLY' in its domain, this set gives the number of legs which the creature has. The value (i.e., the number of legs) associated with the various domain elements can be retrieved by writing F('CAT'), F('MAN'), F('BIRD'), and F('FLY') respectively. If we try to evaluate the value F('DOG'), which is obviously not available in the set F, we get the undefined value OM. The domain of F is the set of all first components of pairs in F, and can be represented by the formula

```
≤PAIR(1), PAIR → F≥
```

All this is shown in the following program.

Program 43

```
LINE STATE
 NO   NO

      /* THE SET OF ALL FIRST COMPONENTS OF PAIRS */
      DO;
         F=≤:<≠CAT≠,4>,<≠MAN≠,2>,<≠BIRD≠,2>,<≠FLY≠,6>≥;
      PRINT,F;
      PRINT,≠THE NUMBER OF LEGS A CAT HAS IS≠,F(≠CAT≠);
      PRINT,≠THE NUMBER OF LEGS A MAN HAS IS≠,F(≠MAN≠);
      PRINT,≠THE NUMBER OF LEGS A BIRD HAS IS≠,F(≠BIRD≠);
      PRINT,≠THE NUMBER OF LEGS A FLY HAS IS≠,F(≠FLY≠);
      PRINT,≠THE DOMAIN OF THE FUNCTION F IS≠,
            ≤PAIR(1),PAIR→F≥;
       COMPUTE; FINISH;
```

Output -- Program 43

```
≤<≠CAT≠ 4> <≠FLY≠ 6> <≠MAN≠ 2> <≠BIRD≠ 2>≥
≠THE NUMBER OF LEGS A CAT HAS IS≠ 4
≠THE NUMBER OF LEGS A MAN HAS IS≠ 2
≠THE NUMBER OF LEGS A BIRD HAS IS≠ 2
≠THE NUMBER OF LEGS A FLY HAS IS≠ 6
≠THE DOMAIN OF THE FUNCTION F IS≠ ≤≠CAT≠ ≠FLY≠ ≠MAN≠ ≠BIRD≠≥

* * * (END OF FILE ON INPUT) * * *
```

The general rule determining the value F(X) for a set F of ordered pairs is this: if F contains precisely one ordered pair P whose first component is X, then F(X) is the second component of P. Otherwise, F(X) is the undefined element OM.

New pairs can be introduced into a set of ordered pairs, and old pairs modified, by a SETLB operation having an outward form very much resembling the "indexed assignment" form familiar from lower level languages such as FORTRAN or ALGOL. Suppose, for example, that we wish to introduce, into the map F considered above, the information that a spider has eight legs, and that a dog has as many legs as a cat. We may write

```
F('SPIDER') = 8;
F('DOG') = F('CAT');
```

This will introduce into F the pairs necessary to record the values implied by the two statements above. This is shown in the following program.

Program 44

```
LINE STATE
 NO    NO

      /* INTRODUCING NEW PAIRS INTO A SET OF ORDERED PAIRS */
      DO;
         F=≤:<≠CAT≠,4>,<≠MAN≠,2>,<≠BIRD≠,2>,<≠FLY≠,6>≥;
      PRINT,F;
      PRINT,≠THE NUMBER OF LEGS A SPIDER HAS IS≠,F(≠SPIDER≠);
         F(≠SPIDER≠)=8; PRINT,F;
         F(≠DOG≠)=F(≠CAT≠); PRINT,F;
      PRINT,≠THE NUMBER OF LEGS A DOG HAS IS≠, F(≠DOG≠);
      COMPUTE; FINISH;
```

Output -- Program 44

```
S<≠CAT≠ 4> <≠FLY≠ 6> <≠MAN≠ 2> <≠BIRD≠ 2>≥
≠THE NUMBER OF LEGS A SPIDER HAS IS≠ OM,
S<≠SPIDER≠ 8> <≠CAT≠ 4> <≠FLY≠ 6> <≠MAN≠ 2> <≠BIRD≠ 2>≥
≤<≠SPIDER≠ 8> <≠CAT≠ 4> <≠DOG≠ 4> <≠FLY≠ 6> <≠MAN≠ 2> <≠BIRD≠
  2>≥
≠THE NUMBER OF LEGS A DOG HAS IS≠ 4

* * * (END OF FILE ON INPUT) * * *
```

A set of ordered pairs can be modified directly by assigning new function values, as in the following program.

Program 45

```
LINE STATE
 NO   NO

      /*  MODIFICATION OF A SET OF ORDERED PAIRS BY
          ASSIGNMENT OF NEW FUNCTION VALUES          */
      DO;
         AGE=≤:<≠SMITH≠,20>,<≠JAMES≠,37>,<≠BROWN≠,53>≥;
      PRINT,AGE;
         AGE(≠SMITH≠) = AGE(≠SMITH≠)+1;
         AGE(≠MOORE≠)=28; PRINT,AGE;
      COMPUTE; FINISH;
```

Output -- Program 45

```
≤<≠BROWN≠ 53> <≠SMITH≠ 20> <≠JAMES≠ 37>≥
≤<≠BROWN≠ 53> <≠SMITH≠ 21> <≠MOORE≠ 28> <≠JAMES≠ 37>≥

* * * (END OF FILE ON INPUT) * * *
```

Let us a ain define a set F:

```
F = ≤:<'CAT',4>,<'MAN',2>,<'BIRD',2>,<'FLY',6>≥
```

Suppose now that we want the inverse map of the function F. SETLB allows us to produce this inverse by using the following statement:

```
HAVELEGS = ≤<X(2),X(1)>, X → F≥;
```

In effect, thsi interchanges the first and second component for each of the tuples X of F. Once this statement has been executed, we may, for example, evaluate

HAVELEGS(6)

getting as its value that creature with 6 legs, i.e., 'FLY'. This is the unique right-hand component of the tuple containing as the first component the value 6.

Writing

HAVELEGS(2)

however, will lead to an indefinite result since there are two tuples in HAVELEGS whose first component is a 2.

All of these features are illustrated in the next program. However, a new concept, which will be explained now, also appears.

By using the set delimiters ≤ ≥ it is possible to obtain the *set of all values* which a multi-valued mapping assumes for a given element of its domain. For example, by evaluating

HAVELEGS≤2≥

we construct the set of all second components of pairs in HAVELEGS of which the number 2 is the first component.

This remark should make every part of the following program clear to the reader.

Program 46

```
LINE STATE
 NO   NO

          /* INVERSE MAP OF FUNCTION */
          DO;
            F=≤;<≠CAT≠,4>,<≠MAN≠,2>,<≠BIRD≠,2>,<≠FLY≠,6>≥;
          PRINT.F;
            HAVELEGS=≤<X(2),X(1)>,X→F≥;
          PRINT.HAVELEGS;
          PRINT.HAVELEGS(6);
          PRINT.HAVELEGS(2);
          PRINT.HAVELEGS≤2≥;
          COMPUTE; FINISH;
```

Output -- Program 46

```
≤<≠CAT≠ 4> <≠FLY≠ 6> <≠MAN≠ 2> <≠BIRD≠ 2>≥
≤<2 ≠BIRD≠> <2 ≠MAN≠> <4 ≠CAT≠> <6 ≠FLY≠>≥
≠FLY≠
OM,
≤≠MAN≠ ≠BIRD≠≥

* * * (END OF FILE ON INPUT) * * *
```

Let us once again make use of the function F introduced above. Define a set S as follows:

S = ≤: 'CAT','MAN','BIRD'≥;

To construct the range of F over the set S, i.e., the set of all values which F assumes for any element of S, all one need write in SETLB is

F[S] .

Note that the set S is enclosed in square brackets. This is shown in the next program.

Program 47

```
LINE STATE
 NO    NO

          /* THE RANGE OF A FUNCTION F ON A SET S */
          DO;
             F=≤:<≠CAT≠,4>,<≠MAN≠,2>,<≠BIRD≠,2>,<≠FLY≠,6>≥;
             S=≤:≠CAT≠,≠MAN≠,≠BIRD≠≥; PRINT,S;
          PRINT,F[S];
          COMPUTE; FINISH;
```

Output -- Program 47

```
≤≠CAT≠ ≠MAN≠ ≠BIRD≠≥
≤2 4≥

* * * (END OF FILE ON INPUT) * * *
```

We have seen in the preceding pages that SETLB allows a set of ordered pairs to be used as a function. In much the same way, a set F of ordered triples may be used as a function of two variables. In this case, F(A,B) designates the third component of the unique triple in F whose first two components are A and B. F may be multivalued, in which case F≤A,B≥ designates the set of all third components of triples on F whose first two components are A and B. New triples may be inserted into F, and components of old triples modified, by assignments of he form F(A,B)=C. Finally, a set of ordered triples may also be used as a (generally multivalued) function of one variable. For example, F≤A≥ designates the set of all tails of triples in F whose head is in A. All this is illustrated by the following program.

Program 48

```
LINE STATE
 NO   NO

            /* A FUNCTION OF SEVERAL VARIABLES */
            DO;
               CHILD=NL.;
               CHILD(≠COHEN≠,1)=≠HORST≠;
               CHILD(≠COHEN≠,2)=≠FATIMA≠;
               CHILD(≠COHEN≠,3)=≠MUGO≠;
            PRINT.CHILD;
            PRINT.CHILD≤≠COHEN≠≥;
               CHILD(≠COHEN≠,3)=≠ELEPHTHERIOS≠;
            PRINT.CHILD;
            COMPUTE; FINISH;
```

Output -- Program 48

```
S<≠COHEN≠ 1 ≠HORST≠> <≠COHEN≠ 2 ≠FATIMA≠> <≠COHEN≠ 3 ≠MUGO≠>≥
S<1 ≠HORST≠> <2 ≠FATIMA≠> <3 ≠MUGO≠>≥
S<≠COHEN≠ 1 ≠HORST≠> <≠COHEN≠ 2 ≠FATIMA≠> <≠COHEN≠ 3 ≠ELEPHTHERIOS≠>≥

* * * (END OF FILE ON INPUT) * * *
```

The following example shows the use of a function of three variables, represented by a set of ordered quadruples. It shows also that the same function can be used as a multivalued function of one and two variables, and even, in cases in which it happens to be single valued, as a single valued function of two variables.

Program 49

```
LINE STATE
 NO   NO

              /* ANOTHER MULTIVALUED FUNCTION */
              DO;
                 CHILD=NL,;
                 CHILD(≠JONES≠,≠GIRL≠,1)=≠MARY≠;
                 CHILD(≠JONES≠,≠BOY≠,2)=≠PETER≠;
                 CHILD(≠JONES≠,≠BOY≠,3)=≠MOISH≠;
              PRINT.CHILD;
              PRINT.CHILD≤≠JONES≠≥;
              PRINT.CHILD≤≠JONES≠,≠BOY≠,2≥;
              PRINT.CHILD(≠JONES≠,≠GIRL≠);
              PRINT.CHILD≤≠JONES≠,≠BOY≠≥;
              COMPUTE; FINISH;
```

Output -- Program 49

```
≤<≠JONES≠ ≠GIRL≠ 1 ≠MARY≠> <≠JONES≠ ≠BOY≠ 2 ≠PETER≠> <≠JONES≠ ≠BOY≠
  3 ≠MOISH≠>≥
≤<≠GIRL≠ 1 ≠MARY≠> <≠BOY≠ 2 ≠PETER≠> <≠BOY≠ 3 ≠MOISH≠>≥
≤≠PETER≠≥
<1 ≠MARY≠>
≤<2 ≠PETER≠> <3 ≠MOISH≠>≥

* * * (END OF FILE ON INPUT) * * *
```

By making the assignment F(A) = OM.;, we remove from the set F all ordered pairs whose first element is A. Similarly, by making the assignment F(A,B) = OM.; we remove from the set F of triples all triples whose first two components are A and B. This important special case is shown in the next example.

Program 50

```
LINE STATE
 NO    NO

             /* REMOVING PAIRS */
             DO;
                F=≤:<≠A≠,1>,<≠B≠,2>,<≠C≠,3>≥;
             PRINT,F;
                F(≠A≠)=OM,;
             PRINT,F;
                F(≠B≠)=OM,;
             PRINT,F;
                G=≤:<5,50,500>,<5,60,550>,<5,70,575>,<6,60,600>,<7,70,700>≥;
             PRINT,G;
                G(5,70)=OM,;
                G(5)=OM,;
             PRINT,G;
                G(6)=OM,;
             PRINT,G;
                G(7,70)=OM,;
             PRINT,G;
             COMPUTE; FINISH;
```

Output -- Program 50

```
≤<≠B≠ 2> <≠A≠ 1> <≠C≠ 3>≥
≤<≠B≠ 2> <≠C≠ 3>≥
≤<≠C≠ 3>≥
≤<5 50 500> <5 60 550> <5 70 575> <6 60 600> <7 70 700>≥
≤<6 60 600> <7 70 700>≥
≤<7 70 700>≥
NL.


* * * (END OF FILE ON INPUT) * * *
```

5.2 An Observation on the Use of Subexpressions within Set Expressions

Let us assign X = 1; Y = 2; and Z = 3. The statement:

A = ≤:X,Y,Z≥;

will form the set {1,2,3}.

If we now modify X, Y and Z by writing:

X = 5; Y = Y + 6; Z = Z - 8;

and print the set A again we find that the set has not altered at all. Only a direct reassignment to the variable A will change its value. The following program illustrates this principle. (Warning: in other more complex and somewhat different circumstances, explained in section 14.5, modification of one set can affect another.)

Program 51

```
LINE STATE
 NO   NO

        /* MORE ON ASSIGNMENTS */
        DO;X=1;Y=2;Z=3;
        A=≤:X,Y',Z≥;PRINT,A;
        X=5;Y=Y+6;Z=Z-8;PRINT,A;
        A=≤:X,Y',Z≥;PRINT,A;COMPUTE;FINISH;
```

Output -- Program 51

```
≤1 2 3≥
≤1 2 3≥
≤8 ¢5 5≥

* * * (END OF FILE ON INPLT) * * *
```

QUESTIONS

Chapter 5

1. Assume the set:

   S = ≤:<'BALL', 'BLACK'>, <'HAT', 'WHITE'>, <'TABLE','RED'>,
   <'BALL', 'GREEN'>≥

   What values result from evaluating the following:

   (a) S('TABLE')
   (b) S('HAT')
   (c) S('CHAIR')
   (d) ≤P(1), P → S≥

2. Assuming the same set S as in the previous question, how will it be affected, if at all, by execution of the following statements in succession?

   (a) S('BALL') = 'BLACK';
   (b) S('BAT') = S('HAT');
   (c) PRINT. S('BAT');
   (d) S('HAT') = OM.;
   (e) PRINT. S;
   (f) S = ≤<K(2), K(1)>, K → S≥;

3. Assume the set:

   SET = ≤:<'VW',4>,<'CORTINA',5>,<'RR',8>,
   <'VV',3>,<'CADDY',8>,<'VW',2>≥;

   What is the result of executing the following SETLB instructions:

   (a) PRINT. SET ≤2≥;
   (b) PRINT. SET ('VW');
   (c) PRINT' SET ≤'VW'≥;
   (d) PRINT. SET ≤'VV'≥;
   (e) PRINT. SET [≤:'VW'≥];
   (f) PRINT. SET [≤:'RR','VV'≥];
   (g) PRINT. SET ('CADDY');
   (h) PRINT. SET ('FORD');

4. Assume the set

```
S = ≤:<'SCHWARTZ', 'ALL ABOUT SETL', 5733>,
                   <'MULLISH', 'ON FORTRAN', 132>,
                   <'LEWIS', ALL ABOUT CAP', 21>,
                   <'SCHWARTZ', MATRIX ALGEBRA', 192>,
                   <' MULLISH', ON PL/I', 160>≥;
```

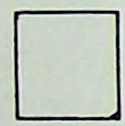
What is the effect of executing the following SETLB instructions:

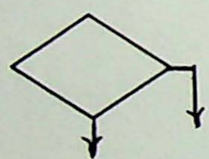
```
(a)      PRINT. S('SCHWARTZ');
(b)      PRINT. S('MULLISH');
(c)      PRINT. S('MULLISH, 'ON PL/I');
(d)      PRINT. S('SCHWARTZ', 'ALL ABOUT SETL');
(e)      PRINT. S≤'SCHWARTZ'≥;
(f)      PRINT. S≤'MULLISH'≥;
```

6. CONTROL STATEMENTS

In each of the programs described so far, statements have been executed sequentially. That is, after each instruction was executed, the following statement was executed, etc. In programming it is often essential to remove this restriction, allowing statements to be executed in a much more flexible, data dependent, order. Statements which accomplish this are called *control statements*. SETLB provides various control features, which we now wish to outline.

In describing the effect of the different SETLB control features, we will occasionally find it helpful to use flow charts. In our flow charts, square boxes will designate blocks of code to be executed in an essentially serial way. Diamond shaped boxes will be used to denote decision points at which tests having one of two possible outcomes will be made. Thus the basic elements in our flowchart vocabulary will be as follows:

(1)  block of code

(2)  decision box
exit path depends on
answer to question in box

These simple elements will be seen to suffice for the representation of almost all the important control features of SETLB.

6.1. <u>Iteration over a Numerical Range</u>

Suppose we set a variable SUM to zero, and then generate the integers 1 to 10 inclusive, adding each integer generated to SUM.

In SETLB these steps may be written:

```
SUM = 0;
(∀ 1 <= N <= 10) SUM = SUM + N;;
```

(The ∀ mark stands for "for all" and is actually the 11-2-8 punch on the 029 keypunch machine.) The result of this operation is the sum of the integers 1 through 10.

The parenthesized "header" sequence appearing in the second line displayed above is an *iterator*.

An iterator serves to repeat the statements (or group of statements) which follow it. The statements which an iterator causes to be repeated are called the *scope* of the iterator. In the example given above, the scope of the iterator (∀ 1 <= N <= 10) is the single statement

```
SUM = SUM + N;
```

The scope of an iterator is ended (or "closed") by a *scope terminating mark*. The simplest form of scope terminating mark is merely an extra semicolon. Note that the second of the two semicolons ends an iteration scope (while the first punctuates a statement). Other, slightly different ways of marking the end of an iterator scope will be explained below. Note that each iterator occurring in a SETLB program requires its own terminator.

The following program illustrates the use of iterators of the simple form just described. It calculates the sum of the squares 1^2 through 10^2. (Of course, this could be done even more easily using a compound operator.) In the seventh line of the following program, the iterator;

(1) (∀ 1 <= N <=↓ TUPLE)

is used to sum the components of the tuple. The necessary addition is performed by the statement:

```
SUM = SUM + TUPLE(N);
```

which is the only statement in the scope of the iterator (1). Note once again that the iteration scope is terminated with a double semicolon.

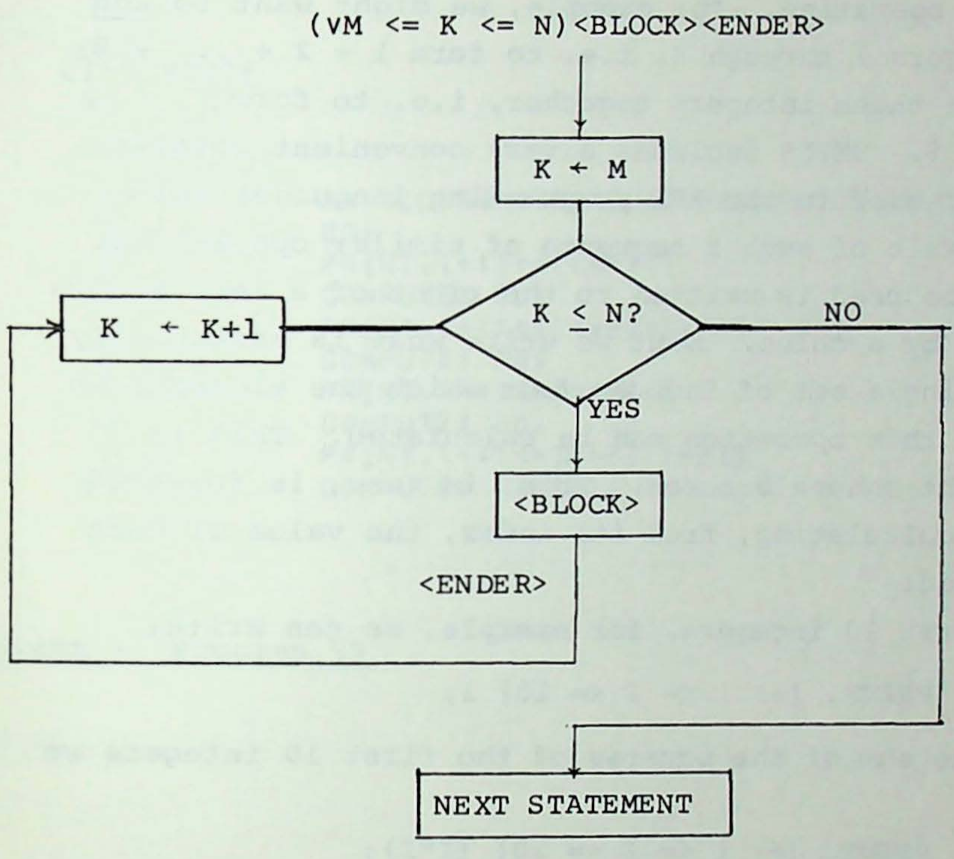
Program 52

```
LINE STATE
 NO    NO

        /* SUMMING */
        DO; SUM=0;
        (∀1<=N<=10)SUM=SUM+N*N;;
        PRINT.SUM,≠IS THE SUM OF THE SQUARES 1 TO 10≠;
        COMPUTE;
        DO; TUPLE=<3,7,8,4>; SUM=0;
        (∀1<=N<=↓TUPLE)SUM=SUM+TUPLE(N);;
        PRINT.SUM,≠IS THE SUM OF THE COMPONENTS≠;
        COMPUTE; FINISH;
```

Output -- Program 52

```
385 ≠IS THE SUM OF THE SQUARES 1 TO 10≠

22 ≠IS THE SUM OF THE COMPONENTS≠

* * * (END OF FILE ON INPUT) * * *
```

The form and meaning of the type of iterator appearing in the above program is shown in the following chart.



6.2. Compound Operators

It is sometimes desirable to combine all the members of a set using some binary operation. For example, we might want to add together the integers 1 through 8, i.e. to form 1 + 2 + ... + 8; or to multiply all these integers together, i.e. to form 1 * 2 * 3 * ... * 8. SETLB includes a very convenient diction (adapted from that used in the APL programming language) for describing the result of such a sequence of similar operations. The operation to be used is written to the right of a left square bracket, followed by a colon. Next we write what is essentially an iterator defining a set of indices from which the elements to be combined using this operation can be calculated. This is followed by a right square bracket. This, in turn, is followed by an expression calculating, from its index, the value of each term to be combined.

To sum the first 10 integers, for example, we can write:

```
PRINT. [+: 1 <= I <= 10] I;
```

To compute the sum of the squares of the first 10 integers we can write:

```
PRINT. [+: 1 <= I <= 10] (I*I);
```

To compute the product of the first 5 integers:

```
PRINT. [*: 1 <= I <= 5] I;
```

To compute the product

(1+3) * (2+3) ... (10+3)

which in standard mathematical notation would be written:

$$\prod_{i=1}^{10} (i + 3)$$

we can write:

```
PRINT.[*: 1 <= I <= 10](I+3);
```

The next program performs various such compuations.

Program 53

```
LINE STATE
 NO   NO

        /* ILLUSTRATIONS OF THE COMPOUND OPERATOR */
        DO;
        PRINT.[+:1<=I<=8]I;
        COMPUTE; DO;
        PRINT.[+:1<=I<=8](I*I);
        COMPUTE; DO;
        PRINT.[*:1<=I<=5]I;
        COMPUTE; DO;
        PRINT.[*:1<=I<=5](I+3);
        COMPUTE; FINISH;
```

Output -- Program 53

```
36

204

120

6720

* * * (END OF FILE ON INPUT) * * *
```

6.2.1. Examples of Set Formers and Compound Operator

6.2.1.1 A Prime Number Generator and The Sum of Primes

To generate the set of prime numbers up to 100 we can use the set former dictions described previously. The program below finds the set of P's between 2 and 100 such that for every N greater than or equal to 2 and less than P the remainder of P/N is not equal to zero. These are the primes.

Program 54

```
LINE STATE
 NO    NO

/* A PRIME NUMBER GENERATOR */
DO;
PRINT.≤P,2<=P<=100↑(∀2<=N<P↑(P//N) NE.0)≥;
COMPUTE; FINISH;
```

Output -- Program 54

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

* * * (END OF FILE ON INPUT) * * *
```

A compound operator may be used to sum up all of the primes below 100. The compactness of the program is striking.

Program 55

```
LINE STATE
 NO   NO

          /* THE SUM OF THE PRIMES BELCW 100 */
          DO;
          PRINT,[+!2<=P<=100+(v2<=N<P+(P//N) NE,0)] P;
          COMPUTE; FINISH;
```

Output -- Program 55

```
1060

* * * (END OF FILE ON INPUT) * * *
```

6.2.1.2 Checking a Formula

It is well known that the sum of the integers 1 to n is given by the formula:

$$\text{Sum} = \frac{n(n+1)}{2}$$

This formula can be checked by a calculation which uses a compound operator. What follows is a program to sum the first 100 integers. The sum is computed by writing:

[+: 1 <= I <= N] I

and the result is compared with that arrived at by evaluating the formula. The following program shows the results to be identical.

The sum of the squares of the integers 1 to n is given by the formula

$$\text{Sum} = n(n+1)(2n+1)/6$$

This too is evaluated in the next program and is checked against the sum computed by writing:

[+: 1 <= I <= N] (I*I)

Care must be taken to completely parenthesize each of the expressions to be compared.

Program 56

```
LINE STATE
 NO   NO

  1          /* CHECKING TWO FORMULAS */
  2          DO; N=10;
  3          PRINT,([+:1<=I<=N] I) EQ,((N*(N+1))/2);
  4          PRINT,([+:1<=I<=N](I*I))EQ,((N*(N+1)*((2*N)+1))/6);
  5          PRINT,([+:1<=I<=N](I*I));
  6          PRINT,((N*(N+1)*((2*N)+1))/6);
  7          COMPUTE; FINISH;
```

Output -- Program 56

```
TRUE
TRUE
385
385


* * * (END OF FILE ON INPUT) * * *
```

6.3. Iteration over the Elements of a Set

The program which follows uses an iteration to sum all the elements of an explicitly given set. The necessary iterator has the form $(\forall X \rightarrow A)$, and causes a block of code to be repeated for each of the elements X of the set A. Note once more that the end of the scope of the iterator $(\forall X \rightarrow A)$ is marked in the simplest way possible, by the presence of an additional semicolon.

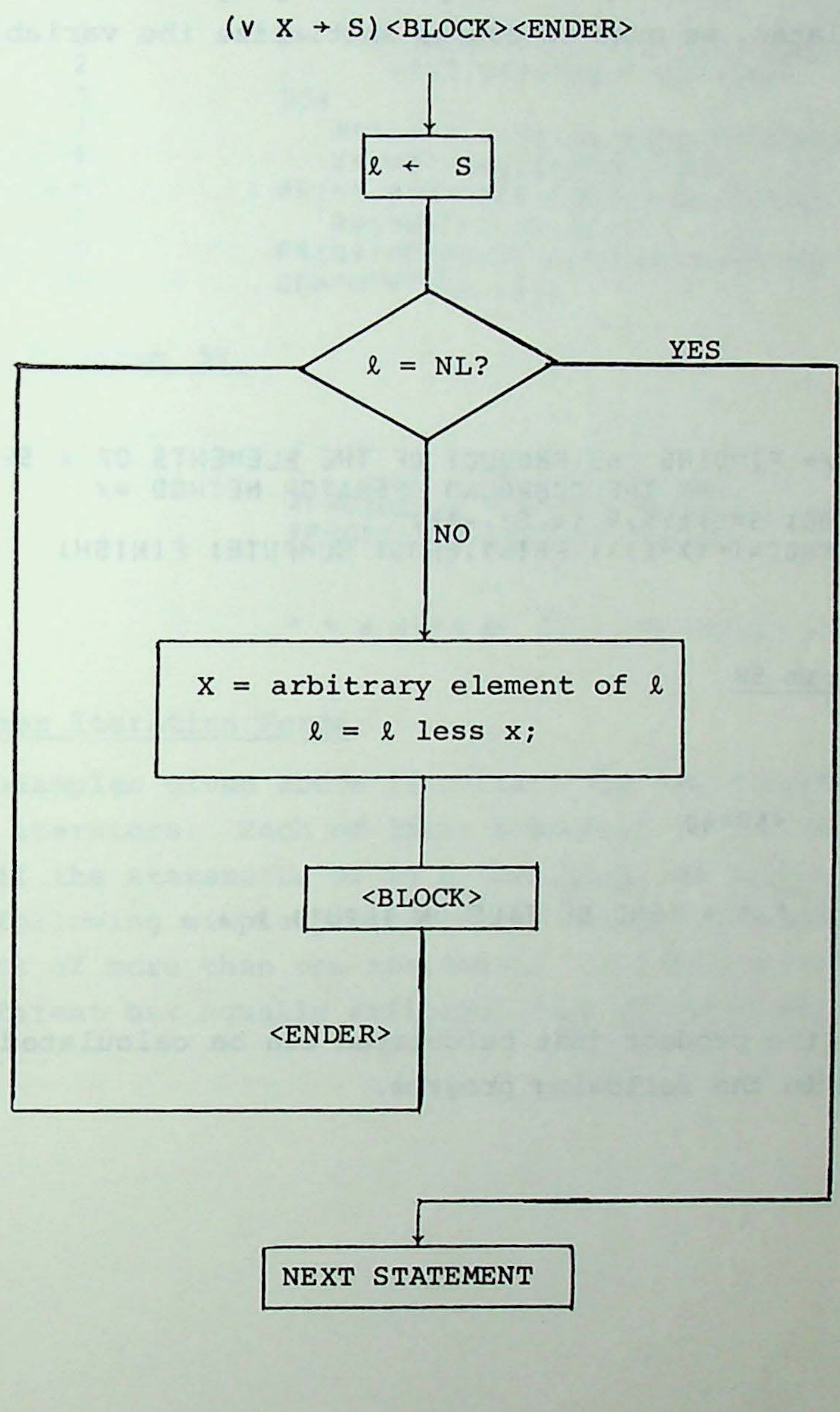
Program 57

```
LINE STATE
 NO   NO

     /* ITERATING AGAIN */
     DO; A=≤:3,7,8,15≥; PRINT,A;
     SUM=0:(∨X→A)SUM=SUM+X:;
     PRINT,≠SUM OF THE ELEMENTS =≠,SUM;
     COMPUTE; FINISH;
```

Output -- Program 57

```
≤8 3 15 7≥
≠SUM OF THE ELEMENTS =≠ 33

* * * (END OF FILE ON INPUT) * * *
```

The form and meaning of the set-theoretic iterator illustrated by the preceding program is shown in the following chart.



In much the same way we may calculate the product of the elements of a set. This is done in the next program using a compound operator as an alternate to the method just described. Should the previous method be preferred, care should be taken to initialize PROD to 1; if, as in the previous program, a sum is to be calculated, we must of course initialize the variable SUM to 0.

<u>Program 58</u>

```
LINE STATE
 NO    NO

        /* FINDING THE PRODUCT OF THE ELEMENTS OF A SET
              BY THE COMPOUND OPERATOR METHOD */
        DO; S=≤:1,5,9,16,21,-4≥;
        PROD=[*:X→S]X; PRINT,PROD; COMPUTE; FINISH;
```

<u>Output -- Program 58</u>

```
-60480

* * * (END OF FILE ON INPUT) * * *
```

Of course, the product just calculated can be calculated both ways, as shown in the following program.

Program 59

```
LINE STATE
 NO   NO

             /* THE PRODUCT OF THE ELEMENTS OF A SET */
             /*     -TWO DIFFERENT METHODS-          */
             DO;
                S=≤:1,5,9,16,21,-4≥; PROD1=1;
                (∀X∈S) PROD1=PROD1*X;;
             PRINT.≠PRODUCT 1 EQUALS≠,PROD1;
                PROD2=[*:X∈S]X;
             PRINT.≠PRODUCT 2 EQUALS≠,PROD2;
             COMPUTE: FINISH;
```

Output -- Program 59

```
≠PRODUCT 1 EQUALS≠ -60480
≠PRODUCT 2 EQUALS≠ -60480

* * * (END OF FILE ON INPUT) * * *
```

6.3. Other Iteration Forms

The examples given above illustrate the two simplest types of SETLB iterators. Each of these iterators can be used to repeat all the statements of an entire group or block.

The following simple program shows how the iterator is used in a block of more than one statement. In addition, it illustrates four different but equally efficient ways of terminating the iterator.

Program 60

```
LINE STATE
 NO    NO

          /* EXAMPLE OF A MULTISTATEMENT BLOCK */
          DO;
             S=≤;5,6,9,-3≥;
          PRINT,S;
             SUM=0; PROD=1;
             (∀X→S) SUM=SUM+X; PROD=PROD*X;;
          PRINT,≠FIRST CALCULATION OF SUM AND PRODUCT≠,SUM,PROD;
             SUM=0; PROD=1;
             (∀X→S) SUM=SUM+X; PROD=PROD*X;END;
          PRINT,≠SECOND CALCULATION OF SUM AND PRODUCT≠,SUM,PROD;
             SUM=0; PROD=1;
             (∀X→S) SUM=SUM+X; PROD=PROD*X;END ∀;
          PRINT,≠THIRD CALCULATION OF SUM AND PRODUCT≠,SUM,PROD;
             SUM=0; PROD=1;
             (∀X→S) SUM=SUM+X; PROD=PROD*X;END ∀X;
          PRINT,≠FOURTH CALCULATION OF SUM AND PRODUCT≠,SUM,PROD;
          COMPUTE; FINISH;
```

Output -- Program 60

```
≤9 -3 5 6≥
≠FIRST CALCULATION OF SUM AND PRODUCT≠ 17 -810
≠SECOND CALCULATION OF SUM AND PRODUCT≠ 17 -810
≠THIRD CALCULATION OF SUM AND PRODUCT≠ 17 -810
≠FOURTH CALCULATION OF SUM AND PRODUCT≠ 17 -810

* * * (END OF FILE ON INPUT) * * *
```

The type of SETLB iterator described in the preceding section, whose meaning is, "for all X in set S repeat block" is written as:

(vX → S) block;

SETLB provides other useful iterator forms. In the first place, one has iterators of the form:

(vM < K <= N) block;

with the meaning "for all K greater than M and not greater than N, repeat block." This is the form used in the next program.

Program 61

```
LINE STATE
 NO    NO

        /* ITERATOR OF THE FORM (vM<K<=N) */
        DO;
           M=1; N=5; TUPL=NULT.;
           (vM<K<=N) TUPL=TUPL+<K>;;
        PRINT.TUPL;
        COMPUTE; FINISH;
```

Output -- Program 61

```
<2 3 4 5>

* * * (END OF FILE ON INPUT) * * *
```

The form,

(∀ M <= K < N) block;

whose meaning should be obvious to the reader, is also provided. This form is used in the following program.

Program 62

```
LINE STATE
 NO   NO

          /* ITERATOR OF THE FORM (∀M<=K<N) */
          DO;
             M=1; N=5; TUPL=NULT.;
             (∀M<=K<N) TUPL=TUPL+<K>;;
          PRINT.TUPL;
          COMPUTE; FINISH;
```

Output -- Program 62

```
<1 2 3 4>

* * * (END OF FILE ON INPUT) * * *
```

A fourth variation on the same theme is the iterator

(∀ M < K < N) block;

This iterator is used in the next example.

Program 63

```
LINE STATE
 NO   NO

        /* ITERATOR OF THE FORM (∀M<K<N) */
        DO;
           M=1; N=5; TUPL=NULT.;
           (∀M<K<N) TUPL=TUPL+<K>;;
        PRINT.TUPL;
        COMPUTE; FINISH;
```

Output -- Program 63

```
<2 3 4>

* * * (END OF FILE ON INPUT) * * *
```

SETLB provides the additional iterator form,

(v M >= K >= N) block;

This causes the repeated execution of block, with K varying from M to N, but in decreasing order. This is illustrated in the following program.

Program 64

```
LINE STATE
 NO   NO

        /* ITERATOR OF THE FORM (∀M>=K>=N) */
        DO;
           M=5; N=1; TUPL=NULT.;
           (∀M>=K>=N)TUPL=TUPL+<K>;;
        PRINT.TUPL;
        COMPUTE; FINISH;
```

Output -- Program 64

```
<5 4 3 2 1>

* * * (END OF FILE ON INPUT) * * *
```

The iterator form

(v M >= K > N) block;

also provides for iteration in decreasing order of values of the parameter K. This is shown in the next program.

Program 65

```
LINE STATE
 NO    NO

          /* ITERATOR OF THE FORM (∨M>=K>N) */
          DO;
          M=5; N=1; TUPL=NULT,;
             (∨M>=K>N) TUPL=TUPL+<K>;;
          PRINT,TUPL;
          COMPUTE; FINISH;
```

Output -- Program 65

```
<5 4 3 2>

* * * (END OF FILE ON INPUT) * * *
```

The iterator form

(v M > K >= N) block;

is illustrated in the next program.

Program 66

```
LINE STATE
 NO    NO

        /* ITERATOR OF THE FORM (∀M>K>=N) */
        DO;
           M=5; N=1; TUPL=NULT.;
           (∀M>K>=N) TUPL=TUPL+<K>;;
        PRINT.TUPL;
        COMPUTE; FINISH;
```

Output -- Program 66

```
<4 3 2 1>

* * * (END OF FILE ON INPUT) * * *
```

An iterator may be combined with a condition to give such forms as

(v M <= K <= N ↑ C(K)) block;

which means "for all K in (M,N) such that C(K) is true, repeat block." This is illustrated in the next program.

Program 67

```
LINE STATE
 NO    NO

        /* ITERATOR OF THE FORM (∀M<=K<=N↑C(K)) */
        DO;
           M=1; N=5; TUPL=NULT.;
           (∀M<=K<=N↑((K*K) GT.10)) TUPL=TUPL+<K>;;
        PRINT.TUPL;
        COMPUTE; FINISH;
```

Output -- Program 67

```
<4 5>

* * * (END OF FILE ON INPUT) * * *
```

Conditions may also be attached to iterators over sets, yielding iteration headers like

$$(\forall\ X \rightarrow S \uparrow C(X))\ \text{block};$$

This header means "for all X in S such that the condition C(X) is met, iterate block." This is illustrated in the next example.

Program 68

```
LINE STATE
 NO    NO

          /* ITERATOR OF THE FORM(∀X→S↑C(X)) */
          DO;
             S=<:1,5,-3,2:>; SUM=0;
             (∀X→S↑(X GT,0)) SUM=SUM+X;;
          PRINT,(SUM EQ,8);
          COMPUTE; FINISH;
```

Output -- Program 68

```
TRUE

* * * (END OF FILE ON INPUT) * * *
```

SETLB allows several iterators to be combined into one compound iterator. This is illustrated by the example

(∀ X → S, M(X) <= K <= N(X) ↑ C(X,K)) block;

To illustrate this general form we define a set S. We let M(X) be X + 1 and let N(X) be X + 4, and define a condition C(X,K) by demanding that the remainder of K divided by X is 1. This compound iteration is shown in the next program.

Program 69

```
LINE STATE
 NO   NO

        /* ITERATOR OF THE FORM (∀X→S,M(X)<=K<=N(X)↑C(X,K)) */
        DO;
           S=≤:1,5,3,4≥; TUPL=NULT,;
           (∀X→S,(X+1)<=K<=(X+4)↑(K//X)EQ.1) TUPL=TUPL+<X>;;
        PRINT,TUPL;
        COMPUTE; FINISH;
```

Output -- Program 69

```
<3 3 4 5>

* * * (END OF FILE ON INPUT) * * *
```

Iterators of all the forms described in the last few pages can also be used in set-formers, existential and universal quantifiers, and compound operators.

6.4. IF, THEN, ELSE

One will often wish to execute portions of a program conditionally, i.e., to execute or not execute a block of code, depending on whether or not a particular Boolean expression evaluates to TRUE. or FALSE. For this purpose, SETLB provides two statment forms: the IF-THEN form and the IF-THEN-ELSE form. The former, and simpler, of these two statement types has the general appearance

IF condition THEN block;

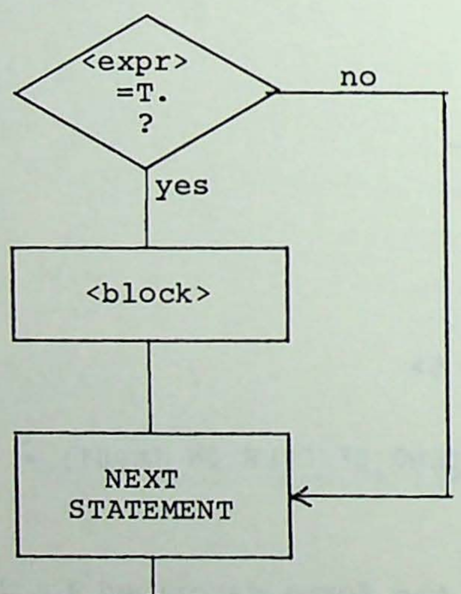
An example would be

IF A EQ. B THEN C = D;;

Suppose that this statement is executed. If the value of A is equal to B, then C is set equal to D. In the event that A does not equal B, C remains unmodified.

The effect of the general IF-THEN statement is shown by the following flow-chart.

IF <expr> THEN <block>;;



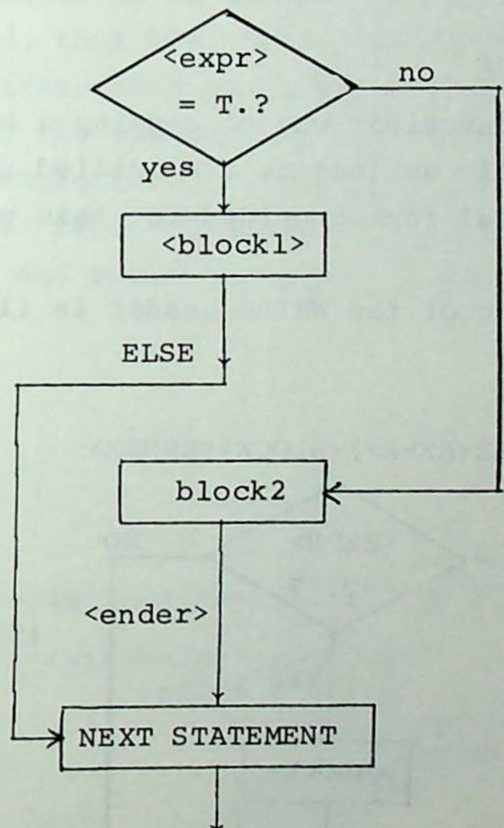
When conditional execution of a block of statements is called for, it will often be the case that one wishes to execute one of two blocks of code, the first if a certain condition is satisfied, the second if the condition fails. For this purpose, SETLB provides the IF-THEN-ELSE statement form. An example of this latter form of conditional statement is as follows:

```
IF E GT. F THEN G = 10;  ELSE G = 15;;
```

In the example shown, G will be set to 10 if E is greater than F. In the contrary case, G will be set to 15. Note that if E is greater than F the block of code introduced by the word ELSE is skipped; i.e. , G = 10; is executed but G=15; bypassed, with control passing to whatever statement follows our sample IF-THEN-ELSE statement.

The general form and significance of an IF-THEN-ELSE statement is shown in the following chart.

IF <expr> THEN <block1>; ELSE <block2>; <ender>



Note that the <ender> marking the end of the scope of an IF-THEN statement, or of an IF-THEN-ELSE statement, can either be a semicolon (leading to the apparent "double semicolon" in the examples shown above), or can be more explicit, as for example, "END IF;".

An "ELSE" can be followed by another "IF," and so on repeatedly, leading to an extended IF-THEN-ELSE structure having the following form:

IF condition (1) THEN block(1); ELSE IF condition (2)
THEN block(2); ...; ELSE block(n);

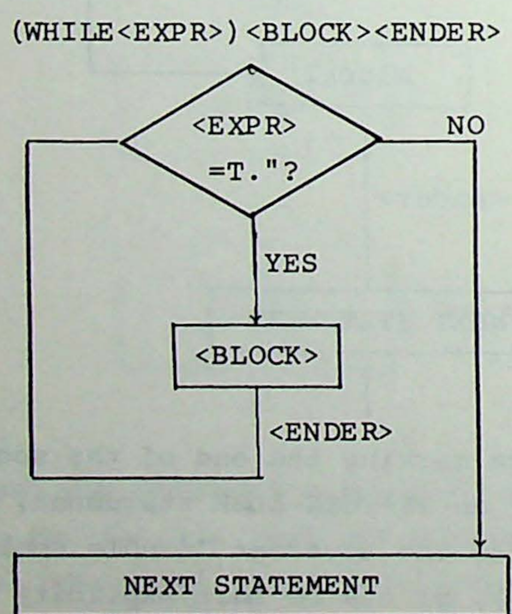
In the above statement block(1) is executed if condition(1) is true, while if condition(2) is true block(2) is executed, etc.

For the moment we postpone giving an example of the SETLB IF-THEN and IF-THEN-ELSE statements. An example will be given in just a few pages, immediately after we have described another useful feature of SETLB, the WHILE iterator.

6.5. The WHILE Iterator

SETLB provides a convenient way of causing a block of statements to be executed repeatedly as long as a specified condition is fulfilled. The dictional form provided for this purpose is the WHILE statement.

The form and meaning of the WHILE header is illustrated by the following chart.



The scope of a WHILE statement may be terminated either by a semicolon, or, as shown in the example below, a more explicit terminator, such as END WHILE; .

The following interesting little program illustrates the use of the WHILE iterator. In it, we apply the following procedure to each of a succession of integers:

(a) if the integer is even, keep halving it until an odd number is reached;

(b) if the number is odd, or becomes odd by repetition of the process (a) above, multiply it by 3, add 1, and repeat (a).

It has been verified experimentally for very many cases that this process, if applied to an integer n, will eventually lead to 1. On the other hand, this has never been proved! We now give a program which verifies this conjecture for all N up to 25, in each case displaying the sequence of intermediate steps passed through before convergence to 1. The reader will note that the expression in line 3 takes advantage of the truncation effect of integer division and associativity of arithmetic to the right.

<u>Program 70</u>

```
LINE STATE
 NO    NO

  1          /* AN ILLUSTRATION OF THE WHILE ITERATOR */
  2          DO;
  3             (v1<=N<=25) ANS=<N>;
  4                (WHILE N NE.1)
  5                   IF(2*N/2)NE.N THEN
  6                           N=(3*N)+1;
  7                   ELSE N=N/2;
  8                   END IF;
  9                ANS=ANS+<N>;
 10                END WHILE;
 11                   PRINT.ANS;
 12          END v1;
 13          COMPUTE; FINISH;
```

Output -- Program 70

```
<1>
<2 1>
<3 10 5 16 8 4 2 1>
<4 2 1>
<5 16 8 4 2 1>
<6 3 10 5 16 8 4 2 1>
<7 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1>
<8 4 2 1>
<9 28 14 7 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1>
<10 5 16 8 4 2 1>
<11 34 17 52 26 13 40 20 10 5 16 8 4 2 1>
<12 6 3 10 5 16 8 4 2 1>
<13 40 20 10 5 16 8 4 2 1>
<14 7 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1>
<15 46 23 70 35 106 53 160 80 40 20 10 5 16 8 4 2 1>
<16 8 4 2 1>
<17 52 26 13 40 20 10 5 16 8 4 2 1>
<18 9 28 14 7 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1>
<19 58 29 88 44 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1>
<20 10 5 16 8 4 2 1>
<21 64 32 16 8 4 2 1>
<22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1>
<23 70 35 106 53 160 80 40 20 10 5 16 8 4 2 1>
<24 12 6 3 10 5 16 8 4 2 1>
<25 76 38 19 58 29 88 44 22 11 34 17 52 26 13 40 20 10 5 16 8
 4 2 1>

* * * (END OF FILE ON INPUT) * * *
```

Indentation was used in the preceding program to make iterator and IF-THEN-ELSE scopes stand out; the standard style of iteration used in the preceding program is emphasized in the following scheme.

```
FOR ALL N
    WHILE N NE. 1
        IF( ) THEN
        ELSE END IF
    END WHILE
    PRINT ANS
END FOR ALL
```

In addition to the WHILE iterator discussed above, whose form is

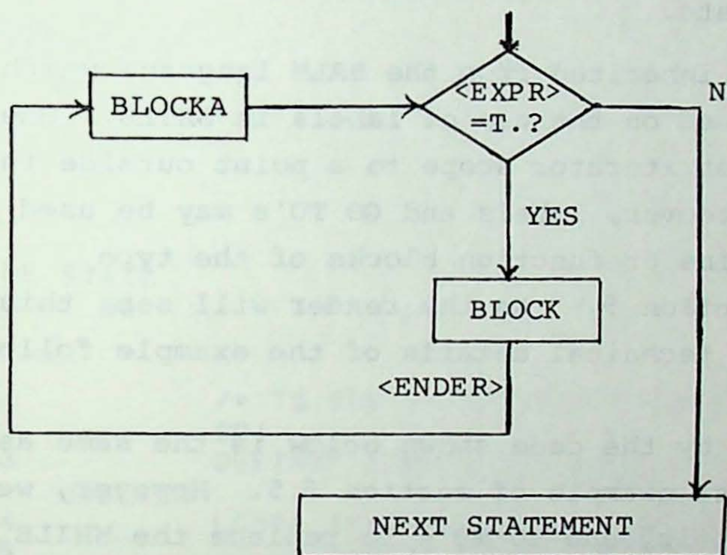
(WHILE G) block;

and which iterates the execution of a block as long as the condition G is fulfilled, SETLB provides a slightly more compound form of the WHILE iterator.

(WHILE G DOING block1) block2;

The meaning of this second WHILE form is expressed clearly by the following flow chart.

(WHILE <EXPR> DOING <BLOCKA>)<BLOCK><ENDER>



The use of this second form of WHILE will be illustrated in some of the longer programs to be given later in the present text.

6.6. Labels and GO-TO Statements

Statements in SETLB programs can be labeled, and control passed to a labeled statement by the use of an explicit transfer or GO TO statement having the form

GO TO label;

Any valid SETL name can be used as a label; labels are designated by the fact that they are followed immediately by a colon. An example of SETLB code containing both a GO TO and a label might be

```
        GO TO loop;
            :
LOOP:   etc.
```

Various restrictions, inherited from the BALM language which underlie SETLB, are imposed on the use of labels in SETLB. One may not jump from within an iterator scope to a point outside this scope, or vice-versa. Moreover, labels and GO TO's may be used only within SETLB subroutine or function blocks of the type discussed in detail in section 9. As the reader will see, this restriction affects a few technical details of the example following below.

The process described by the code shown below is the same as that discussed in the first example of section 6.5. However, we now use IF-statements, labels, and GO TO's to replace the WHILE iterator used in section 7.5. Aside from this technical change, the process carried out remains the same: even numbers are divided by 2 until they become odd; odd numbers are multiplied by 3 and one is added to the result to get an even number. Since, as has just been said, SETLB allows the use of labels only within functions and subroutines, we have been led in the example which follows to define a function ODDEVEN(N) which applies this process to an integer N, building up a tuple which then becomes the function value of ODDEVEN(N). In the example which follows the function is invoked 25 times from within the scope of the iterator (v 1 <= I <= 25). Of the details of the SETLB function definition mechanism, precise discussion of which is postponed to section 10, one only needs to know that

DEFINEF ODDEVEN(N);

introduces the definition of this function; that

END ODDEVEN;

terminates the definition of this function; and that

RETURN ANS;

defines the value of the variable ANS to be also the function value of ODDEVEN(N).

Notice also that the expression N = 3 * N + 1 must be appropriately parenthesized otherwise it will associate to the right i.e. be treated as N = 3 * (N+1) rather than the desired N = (3 * N) + 1.

Program 71

```
LINE STATE
 NO   NO

            /* TO SEE IF THE FIRST 25 INTEGERS ALL GO TO 1 */
            DO;
            DEFINEF ODDEVEN(N); ANS=<N>;
     ODDEVEN
            LOOP: IF(2*(N/2))NE.N THEN GO TO ODD;;
                  N=N/2; ANS=ANS+<N>; GO TO LOOP;
            ODD: IF N EQ. 1 THEN RETURN ANS;
                 ELSE N=(3*N)+1;ANS=ANS+<N>;GO TO LOOP;;
            END ODDEVEN;
            COMPUTE;
            DO;(∀1<=I<=25) PRINT,ODDEVEN(I);; COMPUTE; FINISH;
```

Output -- Program 71

```
<1>
<2 1>
<3 10 5 16 8 4 2 1>
<4 2 1>
<5 16 8 4 2 1>
<6 3 10 5 16 8 4 2 1>
<7 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1>
<8 4 2 1>
<9 28 14 7 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1>
<10 5 16 8 4 2 1>
<11 34 17 52 26 13 40 20 10 5 16 8 4 2 1>
<12 6 3 10 5 16 8 4 2 1>
<13 40 20 10 5 16 8 4 2 1>
<14 7 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1>
<15 46 23 70 35 106 53 160 80 40 20 10 5 16 8 4 2 1>
<16 8 4 2 1>
<17 52 26 13 40 20 10 5 16 8 4 2 1>
<18 9 28 14 7 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1>
<19 58 29 88 44 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1>
<20 10 5 16 8 4 2 1>
<21 64 32 16 8 4 2 1>
<22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1>
<23 70 35 106 53 160 80 40 20 10 5 16 8 4 2 1>
<24 12 6 3 10 5 16 8 4 2 1>
<25 76 38 19 58 29 88 44 22 11 34 17 52 26 13 40 20 10 5 16 8
 4 2 1>

* * * (END OF FILE ON INPUT) * * *
```

6.7. A Remark on Programming Style: GO TO -less Programming.

Long programs become difficult to write, debug, and understand if they become highly cobweb-like, i.e., if each of their parts is actually (or potentially) related to every other part. It is therefore important in approaching a complex programming task to structure it carefully into modules which are as nearly independent of each other as possible. From this point of view, the LABEL-GOTO mechanism is bad, since a labeled statement is potentially related to many other parts of a long program, and since GO-TO's jump about a program in a relatively unrestricted way. Other iterator forms, such as iterators over sets and WHILE iterators, as well as other control mechanisms such as IF-THEN-ELSE statements, can be less dangerous. This suggests that GO TO-free coding represents a desirable style. Believers in this dictum have suggested that the quality of programmers is a decreasing function of the density of GO TO statements they produce and even that the GO TO statement should be abolished from all "higher level" programming languages, perhaps with very limited or special exceptions. SETLB doesn't go to this extreme, but does provide dictions which encourage programmers to be stingy in their use of GO TO's.

6.8. Conditional Expressions.

In SETLB one can write conditional expressions which take on one or another value depending upon whether or not a certain condition has been fulfilled. For example, we can write an assignment involving a conditional expression as follows:

X = IF A GT. B THEN A+B ELSE A*B

When this statement is executed, X will assume the value A+B if A is greater than B and the value A*B otherwise. This conditional expression and others are illustrated by the examples in the next program.

Program 72

```
LINE STATE
 NO    NO

            /* TO ILLUSTRATE THE CONDITIONAL EXPRESSION */
            DO;
               A=1; B=2;
               X=IF A GT. B THEN A*B ELSE A*B;
            PRINT.X;
               A=2; B=1;
            PRINT.IF A GT. B THEN A+B ELSE A-B;
               Y=IF A LT. B THEN A*A*A ELSE B*B*B;
            PRINT.Y;
               A=1; B=2;
            PRINT.IF A LT. B THEN A*A*A ELSE B*B*B;
            COMPUTE; FINISH;
```

Output -- Program 72

```
2
3
1
1

* * * (END OF FILE ON INPUT) * * *
```

Actually, this conditional SETL expression is a more sophisticated facility than the preceding program might suggest. Indeed, continued ELSE IF clauses are allowed within a conditional expression. This is illustrated in the next program. In this program the result of a first IF test is false, so the following ELSE is examined. But as this ELSE is trailed by another IF, also with a false condition, a third ELSE is examined. This also is followed by an IF, this time with a TRUE condition. Consequently, A-B is assigned to X, an outcome which is confirmed in the printed output.

Program 73

```
LINE STATE
 NO   NO

        /* FURTHER ILLUSTRATION OF THE CONDITIONAL EXPRESSION */
        DO;
            A=1; B=2; C=3;
            X=IF A NE.1 THEN B/A ELSE IF C EQ.4 THEN
            A/B ELSE IF C EQ.(A+B) THEN A+B ELSE A-B;
        PRINT.X;
        COMPUTE; FINISH;
```

Output -- Program 73

```
3

* * * (END OF FILE ON INPUT) * * *
```

The reader will no doubt recall that in section 6.4 we illustrated the use of the IF-THEN-ELSE statement, and of the simpler IF-THEN statement, both of which are available in SETLB. In the conditional expression case however, an expression of the form IF-THEN without an ELSE is logically meaningless and will be flagged as an error if used.

Note also that a statement containing a conditional expression is terminated by a simple semicolon rather than by a pair of semicolons.

QUESTIONS

Chapter 6

1. Using a compound operator, write a SETLB instruction to compute the sum of the first 100 integers.
2. Using two compound operators, write a single instruction to compute the difference between the sum of the first 50 integers and the product of the first 10 integers.
3. Write a single instruction to compute the sum of the integers between 1 and 100 which are exactly divisible by 3.
4. Write a single instruction to compute the sum of all the odd numbers from 1 to 101.
5. Write a single instruction to compute the product of the prime numbers from 1 to 20.
6. The square of a number N is equal to the sum of the first N odd integers. Write a program to prove this for N = 20.
7. The <u>factorial</u> of a positive number n is defined as

   $$n * (n-1) * (n-2) * \ldots * 1$$

   Write a single SETLB instruction to compute the factorial of 8.
8. The number of ways that n objects may be taken m at a time is given by the formula:

   $$\binom{n}{m} = \frac{n!}{m!(n-m)!}$$

   Write a SETLB instruction to compute the number of ways 6 objects may be taken 2 at a time.
9. Use a SETLB iterator to compute the following values:
   (a) the sum of the first 50 integers
   (b) the product of the first 5 integers
   (c) the sum of the squares of the first 10 integers
   (d) the sum of the factorials 1 through 5
   (e) the sum of the first 10 odd numbers

10. (a) Assume the set S:

S = ≤:5,9,-2,4,6≥;

Compute the sum of the elements of S by using (a) a single iterator, and (b) a compound operator.

(b) Assuming the same set S as above, compute the product of its elements, again using both an iterator and a compound operator.

11. Using a single iterator compute the tuple whose components consist of
    (a) the first 10 integers
    (b) the first 10 odd integers

    Do this also using a compound operator.

12. What output is generated by the following program (assume that the set S is stored as defined).

```
DO;
   S = ≤:3,9,-2,-1,4≥;  TUP = NULT.;  SUM = 0;
   (∀ X → S ↑ X GT. 0) TUP = TUP + <X>;;
   PRINT. TUP;
   (∀ X → S ↑ X LT. 0) SUM = SUM + X;;
   PRINT. SUM;
COMPUTE;  FINISH;
```

13. What will be the result printed by the following program.

```
DO;
   A=1;  B = 2; C = 3;
   IF (A+B) GT. C THEN D = A*B-C;
   ELSE D = C*B-A;  END IF;
   PRINT. D;
COMPUTE;  FINISH;
```

7. CHARACTER STRINGS

SETLB allows one to deal with data items which are *character strings*, as distinct from tuples, or sets, or numerical values. A character string is an ordered sequence of characters. In its external representation, it begins with a quote and ends with a quote, Between the quotes one may place any keypunch character whatever, including, of course, the blank space, but excluding the quote sign itself.

Suppose we define a character string, say "HENRY" and call it WORD. To determine how many characters are present in the string we merely use the "enumeration" operator, represented, as before, by the downward arrow.

```
WORD = "HENRY";
NWRD = ↓WORD;
```

These operations occur at the start of the following simple program, which shows how an iterator can be used to reverse the order of the characters in a simple string. In the program, character string concatenation, representation by the sign '+', is used; this operation is explained in more detail in comments which follow the program. The "null character string," which consists of zero characters, is denoted in SETLB as NULC.

Program 74 LINE NO STATE NO

```
/* REVERSING A CHARACTER STRING */
DO;
    WORD=≠HENRY≠;
    NWORD=↓WORD;
    NEW=NULC,;
        (∨1<=N<=NWORD) NEW=NEW+WORD(NWORD+1-N);
        END ∨;
PRINT,WORD;
PRINT,NEW:
COMPUTE; FINISH;
```

Output -- Program 74

```
≠HENRY≠
≠YRNEH≠


* * * (END OF FILE ON INPUT) * * *
```

In the preceding program, the variable NEW is set equal to the null character string. On the first iteration of the statement NEW = NEW + WORD(NWORD+1-N), N is equal to 1 and NEW becomes NULC., the null character string, concatenated with WORD(NWORD+1-N), where NWORD = 5. Therefore, NEW is set equal to WORD(5+1-1)=WORD(5), which is the letter Y.

Next, with N = 2, NEW is set equal to the concatenation of NEW, which is "Y", plus WORD(5+1-2), which is WORD(4). This is, of course, "R", and it is concatenated to "Y" to give "YR". This process continues, ultimately building up the desired result: a string within which the characters of the original character string occur in reverse order.

The program found below gives another simple example of character string processing. An English phrase, enclosed within quote signs is assigned as the value of the variable A. The program then removes all the blanks from the phrase.

We let N be the total number of characters in the phrase and initialize C to be the null character string.

Using an iterator which bypasses all blank characters in the string we concatenate each nonblank character to C. This builds up the desired output.

<u>Program 75</u>

```
LINE STATE
 NO   NO

          /* SQUEEZING OUT BLANKS FROM A CHARACTER STRING */
          DO;
          A=≠A DOG HAS FOUR LEGS≠;
          N=♦A; C=NULC.;
          (~1<=I<=N ♦ A(I) NE.≠ ≠) C=C+A(I);;
          PRINT.C; COMPUTE; FINISH;
```

<u>Output -- Program 7.5</u>

```
≠ADOGHASFOURLEGS≠

* * * (END OF FILE ON INPUT) * * *
```

In much the same way we can eliminate any character or set of characters from a character string. In the following program we eliminate all the vowels from a string, using a compound operator rather than an iterator.

Program 76

```
LINE STATE
 NO   NO

/* TO DEVOWELIZE A STRING */
DO;
A=≠THE CAT JUMPED OVER THE BRIGHT MOON≠;
N=#A; C=NULC,;
VOW=≤:≠A≠,≠E≠,≠I≠,≠O≠,≠U≠≥;
PRINT,[+:1<=I<=N↑NOT.A(I)→VOW] A(I);
COMPUTE; FINISH;
```

Output -- Program 76

```
≠TH CT JMPD VR TH BRGHT MN≠

* * * (END OF FILE ON INPUT) * * *
```

7.1. Substrings

SETLB allows one to extract any part of a character string -- the extracted part is, of course, a string. For example, if:

A = 'HOTDOG';

then by writing B = A(1:3) we extract the substring B which begins with the first character of string A and which is three characters long. Observe that the numbers 1 and 3 are separated by a colon and placed within parentheses. The value of B would therefore be 'HOT'. In the same way C = A(4:3); would assign the value 'DOG' to C.

When extracting substrings, one cannot use a negative number on either side of the colon.

The next program illustrates the extraction of substrings.

Program 77

```
LINE STATE
 NO    NO

          /* SOME SUBSTRINGS */
          DO; A=≠APOLLO 17 IS THE BEGINNING RATHER THAN THE END≠;
          PRINT.A;PRINT,A(8:2);PRINT,A(18:3);
          PRINT.A(1:↓A);
          COMPUTE; FINISH;
```

Output -- Program 77

```
≠APOLLO 17 IS THE BEGINNING RATHER THAN THE END≠
≠17≠
≠BEG≠
≠APOLLO 17 IS THE BEGINNING RATHER THAN THE END≠

* * * (END OF FILE ON INPUT) * * *
```

QUESTIONS

Chapter 7

1. Assume the character string:

   S = 'LOVE BLOOMS AT NIGHT';

   What will be printed by the following instruction?

   (a) PRINT. ↓S;

   (b) PRINT. 'I' + S(1:5) + 'DW' + S(17:4);

2. What is printed by the following instructions:

   A = 'T'; B = 'B'; C = 'S'; D = 'L'; E = 'E';

   LANGUAGE = C + E + A + D + B;

   PRINT. LANGUAGE;

3. Assume the character string:

   STR = 'THE CHIRPING BIRDS WORK AND FLIRT';

   Write a program to replace every occurrence of 'IR' or 'OR' with 'OI' and print out the amended string.

4. Assume a character string and compute the total number of vowels therein.

5. Assume a character string and compute the number of letters in it which are vowels and the number which are consonants, ignoring blanks.

6. Assume the character string

   S = 'MOONLIGHT BECOMES YOU';

   Write a program to alphabetize the string, squeezing out the intervening blanks.

7. Write a program to reverse the order of the characters in a character string.

8. Set up a mapping M which sends each letter of the alphabet into its Morse code representation (for example, M('S') = '...'; M('O') = '---'). Using this mapping write a program to convert arbitrary sentences into Morse code, and another program to perform the reverse conversion.

8. MORE EXAMPLES OF THE USE OF SETLB

8.1 A Sorting Algorithm

Let a tuple of numbers, as for example

TUPL = <2,3,9,8,-6,4,5>

be given.

It is often desirable to sort such a sequence of numbers, placing them, let us say, into descending order. We shall now give a SETLB program which does just this. The sort program to be given will illustrate the use of the SETLB iterator and the existential form.

The following remarks will help the reader to understand the program. It sorts by the so-called "exchange" method. Specifically, it searches from left to right through the components of a tuple, looking for components out of order. If an out-of-order pair is found, an interchange is performed. Here is the code.

Program 78

```
        /* AN EXCHANGE SORT */
        DO;
        TUPL=<2,3,9,8,-6,4,5>;
        PRINT.TUPL;
        (WHILE ∃1<=N<↓TUPL↑TUPL(N)LT.TUPL(N+1))
                KEEP=TUPL(N); TUPL(N)=TUPL(N+1);TUPL(N+1)=KEEP;
        END WHILE;
        PRINT.TUPL;
        COMPUTE; FINISH;
```

Output -- Program 78

```
<2 3 9 8 -6 4 5>
<9 8 5 4 3 2 -6>


* * * (END OF FILE ON INPUT) * * *

EXECUTION TIME      2,570 SECONDS
```

8.2. Counting Character Frequencies

In the following example we use an iterator to count the number of times each character appears in a character string. The resulting frequencies are then printed out in a sorted tabular format. Sorting is accomplished by a method resembling that used in the preceding example.

Program 79

```
/* FREQUENCY COUNTER FOR APPEARANCE OF CHARACTERS */
DO;

STRING=≠IN THE FOLLOWING EXAMPLE WE USE AN ITERATOR≠;
PRINT.STRING;

FREQ=NL,; /* FREQUENCY FUNCTION INITIALLY NOWHERE DEFINED */
(∀1<=N<=#STRING)
        C=STRING(N);
        IF FREQ(C) EQ.OM.THEN /* A NEW CHARACTER IS SEEN */
             FREQ(C)=1; /* SINCE FIRST OCCURRENCE */
        ELSE
             FREQ(C)=FREQ(C)+1; /* INCREMENT OCCURRENCE NUMBER */
        END IF;
END ∀;

/*NOW MAKE UP A SEQUENCE CONTAINING ALL CHARACTERS */
SEQ=[+:X∈FREQ]<X(1)>;
/* PERFORM INTERCHANGE WHENEVER A LOWER FREQUENCY CHARACTER PRECEDES */
    /* A HIGHER FREQUENCY CHARACTER */
(WHILE ∃1<=N<#SEQ|FREQ(SEQ(N))LT.FREQ(SEQ(N+1)))
        X=SEQ(N); SEQ(N)=SEQ(N+1);SEQ(N+1)=X;
END WHILE;

PRINT.≠THE CHARACTERS OCCURRING ARE:≠,SEQ,≠. THEIR FREQUENCIES ARE:≠;
/*  NOW PRINT OUT SUCCESSIVE LINES OF TABLE */
(∀1<=I<=#SEQ) PRINT.SEQ(I),FREQ(SEQ(I));;

COMPUTE; FINISH;
```

Output -- Program 79

```
≠IN THE FOLLOWING EXAMPLE WE USE AN ITERATOR≠
≠THE CHARACTERS OCCURRING ARE:≠ <≠ ≠ ≠E≠ ≠T≠ ≠A≠ ≠N≠ ≠O≠ ≠L≠ ≠I≠ ≠W≠ ≠R
 ≠ ≠G≠ ≠H≠ ≠X≠ ≠U≠ ≠F≠ ≠S≠ ≠P≠ ≠M≠> ≠, THEIR FREQUENCIES ARE:≠
≠ ≠ 7
≠E≠ 6
≠T≠ 3
≠A≠ 3
≠N≠ 3
≠O≠ 3
≠L≠ 3
≠I≠ 3
≠W≠ 2
≠R≠ 2
≠G≠ 1
≠H≠ 1
≠X≠ 1
≠U≠ 1
≠F≠ 1
≠S≠ 1
≠P≠ 1
≠M≠ 1

* * * (END OF FILE ON INPUT) * * *
```

9. SUBPROGRAMS

It is very often desirable to apply some one particular process P to several different data items or data structures occurring at several points within some long program. For this to be done without it becoming necessary to repeat the code defining the process P, some mechanism of "detour and return" with transmission of arguments and return of calculated values is required. For this reason, all seriously intended programming languages contain procedure definition mechanisms. SETLB is no exception. It provides facilities for defining procedures of two types: subroutines and functions. In the present section we will explain the conventions which allow subprograms of these two kinds to be defined and used.

9.1. User-Defined Functions

A SETLB function is a subprogram and is introduced by the word DEFINEF. This keyword is followed by the function name, which it is up to the programmer to specify. Names of parameters which are to be transmitted between the main program and the function subprocess are enclosed within parentheses. A function subprogram always computes and returns a value, its so-called function value. This value is returned by executing a RETURN statement. Such a statement consists of the word RETURN followed by an expression. The value of this expression is the value returned by the function subprogram. The whole body of a function subprogram is terminated by an END statement, consisting of the keyword END followed by the name of the function being ended. A SETLB subprogram should generally be defined before it is invoked.

An example will make the meaning of these generalizations plain. Suppose that as part of some larger process we will often have occasion to calculate the sum of the squares of two variables. For this purpose, we may well wish to define a function SUMOFSQ, with two parameters, which calculates and returns a value equal to the sum of the squares of these two parameters. Once this is done, we may, for example, write

```
C = SUMOFSQ(A,B);
```

This will have the same force as the explicit C = (A*A)+ (B*B).

The overall features of the SETLB function-definition mechanism are illustrated in the following example, which is presented to exemplify principles rather than for its practical significance.

Notice that the name of a defined function is automatically printed out by the compiler on the left-hand side of the line following the word 'DEFINEF'; this line is <u>not</u> part of the original program.

<u>Program 80</u>

```
LINE STATE
 NO   NO

  1              /* AN EXAMPLE OF A FUNCTION */
  2              DO;
  3              DEFINEF SUMOFSQ(A,B);
     SUMOFSQ
  4                       RETURN (A*A)+(B*B);
  5                       END SUMOFSQ;
  6              COMPUTE;
  7              DO;
  8                 A=3; B=5; C=SUMOFSQ(A,B);
  9              PRINT,C; COMPUTE; FINISH;
```

<u>Output -- Program 80</u>

```
               34

               * * * (END OF FILE ON INPUT) * * *
```

9.2. Subroutines

Another important type of subprocess is the subroutine. Subroutines behave much like function subprograms; however, they do not return a value, but instead are executed for their effect on the values of existing program variables. Subroutines are introduced by the word DEFINE, which is followed by the name of the subroutine, and then by a parenthesized list of names designating the arguments to be transmitted to the subroutine. When a subroutine has finished its work, control is returned to the program that called the subroutine by executing a RETURN statement. As distinct from a RETURN statement in a function-type subprocedure, a RETURN statement in a subroutine has simply the form

RETURN;

with no expression following it.

In our next example parameter values X and Y are simply printed out by subroutine RETORT; this again is merely illustrative. Notice that the name of the subroutine is automatically printed out on the line following the keyword 'DEFINE'; this work is not part of the original program.

Program 81

```
LINE STATE
 NO    NO

            /* AN EXAMPLE OF A SUBROUTINE */
            DO;
               DEFINE RETORT(X,Y);
       RETORT
            PRINT,≠X=≠,X,≠AND Y=≠,Y;
               RETURN;
               END RETORT; COMPUTE;
            DO;
               X=1; Y=2;
               RETORT(X,Y);
            COMPUTE; FINISH;
```

Output -- Program 81

```
≠X=≠ 1 ≠AND Y=≠ 2

* * * (END OF FILE ON INPUT) * * *
```

What follows is another simple example of a main routine which uses a single function-type subroutine. The function merely calculates the maximum of its two parameter values.

Program 82

```
LINE STATE
 NO   NO

             /* A USER DEFINED -MAXIMUM- FUNCTION */
             DO;
                DEFINEF MYMAX(A,B);
      MYMAX
                   IF A GT. B THEN RETURN A;
                              ELSE RETURN B;;
                END MYMAX;
             COMPUTE; DO;
             A=1; B=2;
             PRINT,≠THE MAXIMUM OF≠, A, ≠AND≠, B,≠IS≠,MYMAX(A,B);
             COMPUTE; FINISH;
```

Output -- Program 82

```
≠THE MAXIMUM OF≠ 1 ≠AND≠ 2 ≠IS≠ 2

* * * (END OF FILE ON INPUT) * * *
```

There is no limit to how many subprograms a program may have. Furthermore, a program can use both function and subroutine subprograms. This is illustrated in the next program. In the main section of this program X is set to 7 and Y to 2. The function CALC is invoked and it computes the sum of the remainder of A/B and B/A. The result is printed by the subroutine STATE.

Program 83

```
LINE STATE
 NO    NO

        /* AN EXAMPLE OF A PROGRAM INVOLVING
        BOTH A SUBFUNCTION AND A SUBROUTINE */
        DO; DEFINEF CALC(A,B);RETURN(A//B + B/A); END CALC; COMPUTE;
   CALC
        DO; DEFINE STATE(C); PRINT,≠THE RESULT IS≠,C;RETURN; END STATE;COMPUTE;
  STATE
        DO; X=7; Y=2;
        STATE(CALC(X,Y));
        COMPUTE; FINISH;
```

Output -- Program 83

```
≠THE RESULT IS≠ 1

* * * (END OF FILE ON INPUT) * * *
```

We shall now give an example showing how the exchange sorting procedure described in section 8.1 can be incorporated into a subroutine and thereby made useable at many points in a main program. In what follows, SORT is the subroutine, which is called twice in the main program which follows it.

Program 84

```
LINE STATE
 NO   NO

          /* A SORTING ROUTINE */
          DO;
          DEFINE SORT(TUPL);
     SORT
          (WHILE ∃1<=N<#TUPL↑TUPL(N) GT.TUPL(N+1))
             KEEP=TUPL(N); TUPL(N)=TUPL(N+1); TUPL(N+1)=KEEP;
          END WHILE;
          RETURN;
          END SORT;

          COMPUTE; DO;
          TUPL=<2,3,9,8,-6,4,5>;
          PRINT.TUPL; SORT(TUPL);PRINT.TUPL;
          TUPL=<-1,-2,-3,1,2,3>;
          PRINT.TUPL; SORT(TUPL); PRINT.TUPL;
          COMPUTE; FINISH;
```

Output -- Program 84

```
<2 3 9 8 -6 4 5>
<-6 2 3 4 5 8 9>
<-1 -2 -3 1 2 3>
<-3 -2 -1 1 2 3>

* * * (END OF FILE ON INPUT) * * *
```

Next we give a lexicographic comparison function, i.e., a function which compares character string lexicographically, deciding which of the two strings would appear first in standard "dictionary order."

Program 85

```
LINE STATE
 NO    NO

        /* A LEXICOGRAPHIC COMPARISON ROUTINE */
        DO;

        DEFINEF ALPHBIGR(A,B); /* DECIDES WHETHER B SHOULD FOLLOW A */
   ALPHBIGR
                               /* DEFINE ALPHABETIC ORDER OF CHARACTERS BY A  */
                                  /* STRING AND BY A MAP                       */
        CHARSTR=≠ABCDEFGHIJKLMNOPQRSTUVWXYZ≠;
        CHARPOS=≤<CHARSTR(N),N>,1<=N<=↓CHARSTR≥;
                               /* FIND MINIMUM OF ARGUMENT LENGTHS             */
        IF(↓A) GT.↓B THEN LMIN=↓B;
                  ELSE LMIN=↓A;;
                               /* SEEK FIRST CHARACTER IN WHICH A AND B DIFFER */
        IF ∃1<=N<=LMIN↑ B(N) NE.A(N) THEN
                               /* FIND WHICH COMES FIRST IN CHARSTR            */
              IF CHARPOS(B(N)) GT. CHARPOS(A(N)) THEN
                  RETURN T.;
              ELSE
                  RETURN F.;
              END IF CHARPOS;
        ELSE /* IF THE SHORTER OF A AND B IS IDENTICAL WITH THE FIRST PART    */
                          /* OF THE OTHER */
        IF(↓A)LE. ↓B THEN RETURN T.;
        ELSE RETURN F.;
        END IF ∃;
        END ALPHBIGR;

        COMPUTE; DO;

        PRINT.≠CAT COMES AFTER COT IS≠,ALPHBIGR(≠COT≠,≠CAT≠);
        PRINT.≠CABBAGE COMES AFTER CAB IS≠,ALPHBIGR(≠CAB≠,≠CABBAGE≠);
        PRINT.≠DOG COMES AFTER DOGWOOD IS≠,ALPHBIGR(≠DOGWOOD≠,≠DOG≠);

        COMPUTE; FINISH;
```

Output -- Program 85

```
≠CAT COMES AFTER COT IS≠ FALSE
≠CABBAGE COMES AFTER CAB IS≠ TRUE
≠DOG COMES AFTER DOGWOOD IS≠ FALSE

* * * (END OF FILE ON INPUT) * * *
```

QUESTIONS

Chapter 9

1. Write a function subprogram which converts a tuple of integers into another tuple in which each component is squared.
2. Write a function subprogram which returns the arithmetic mean of two variables.
3. Write a function subprogram which returns one-third of the cube of a number.
4. Write a subroutine which prints out the components of a tuple in reverse order.
5. Write a subroutine which prints out the number of components of a given tuple and the value of the minimum component of the tuple.
6. Write a function subprogram which, given a character string S made up of words separated by blanks, will return a tuple consisting of the words of S arranged in alphabetical order.
7. Write a function subprogram which, given an integer N, will return the set of all primes whose squares divide N.
8. Write a function subroutine which is able to classify twenty names as being either 'girls' names' or 'boys' names'. Can you make the program handle such names as 'MARYANN', 'JOANNA', 'PAULA', 'LINDA', etc. without adding substantially to the size of the program required?

10. BUILT-IN FUNCTIONS AND OPERATORS PROVIDED BY SETLB

10.1 Absolute Value Operator

The absolute value of a number can be found by using the absolute value operator ABS. , followed by its argument, as in the the next program.

Program 86

```
LINE STATE
 NO    NO

         /* THE ABSOLUTE VALUE FUNCTION */
         DO; A=1; B=-1;
         PRINT.A EQ.ABS.B;
         COMPUTE; FINISH;
```

Output -- Program 86

```
TRUE

* * * (END OF FILE ON INPUT) * * *
```

10.2. The Maximum and the Minimum Operators

The maximum or minimum of two numbers is found by the two dyadic operators MAX. and MIN. , both of which are part of the SETLB library. The next program is self-explanatory.

Program 87

```
LINE STATE
 NO    NO

             /* FINDING THE MAXIMUM AND MINIMUM */
             DO; A=10; B=20;
             PRINT. A MAX. B, ≠= MAXIMUM≠;
             PRINT. A MIN. B , ≠= MINIMUM≠;
             COMPUTE; FINISH;
```

Output -- Program 87

```
20 ≠= MAXIMUM≠
10 ≠= MINIMUM≠

* * * (END OF FILE ON INPUT) * * *
```

The next program shows that the MIN. operator may also be used in a compound operator. First a tuple T is defined. The smallest component is then found by determining the minimum of the first and second components, and then the minimum of this result with the third component, etc. This process continues until every component has been examined.

Any binary operator, including user defined binary operators which are explained in section 9.1, may be used in the SETLB compound operators construction.

<u>Program 88</u>

```
LINE STATE
 NO    NO

       /* USING MIN, AS A COMPOUND CPERATOR */
       DO;
       T=<1,5,6,-2,-8,0,-3,-2//3,-7001>;
       PRINT, [MIN,[1<=N<=#T] T(N));
       COMPUTE; FINISH;
```

<u>Output -- Program 88</u>

```
-7001

* * * (END OF FILE ON INPUT) * * *
```

10.3. The Random Function

Random numbers are available in SETLB via the built-in RANDOM function. The argument of RANDOM is an integer N ; the value returned will be an integer selected at random from 1 to N. This is illustrated in the next program where the roll of two dice is simulated for 10 rolls.

Program 89

```
LINE STATE
 NO   NO

          /* THE RANDOM FUNCTION */
          DO;
          (∀1<=I<=10)
             A=RANDOM(6); B=RANDOM(6);
             Y=A+B;
             PRINT,A,B,Y;
          END ∀1;
          COMPUTE; FINISH;
```

Output -- Program 89

```
1 4 5
1 2 3
2 4 6
1 2 3
1 5 6
1 3 4
5 1 6
5 4 9
1 4 5
1 2 3

* * * (END OF FILE ON INPUT) * * *
```

10.4. IN. and OUT.

Elements may be added to or deleted from a set by use of the IN. and OUT. operators. The statement A IN. B; has the meaning B = B WITH. A; the statement A OUT. B; has the meaning B = B LESS. A; . This is illustrated in the following program.

Program 90

```
LINE STATE
 NO    NO

        /* IN AND OUT */
        DO;
          A=≤:1,2,3≥; B=4; C=3;
        PRINT,A;
          B IN. A;
        PRINT,A;
          C OUT.A;
        PRINT,A;
        COMPUTE; FINISH;
```

Output -- Program 90

```
≤1 2 3≥
≤1 2 3 4≥
≤1 2 4≥

* * * (END OF FILE ON INPUT) * * *
```

The IN. operator is used again in the following program, which converts a character string to a set. The program also illustrates the use of the "DEC." operator, which converts a number to its decimal string representation, and vice-versa. Specifically, the literal quantity "25" is defined. When printed out, it appears not as an integer, but as a character string. However, when this string is acted on by DEC. and printed, the integer equivalent, i.e. 25, appears. Next, the character string "0123456789" is assigned as a value to the variable DIGITUP while DIGITS is initialized to the null set. An iteration then generates the set of all the individual "characters" of DIGITUP.

<u>Program 91</u>

```
LINE STATE
 NO    NO

/* CONVERTING A CHARACTER STRING TO A SET */
DO;
QNUM=≠25≠; PRINT,≠QNUM= ≠,QNUM;
NUM=DEC,QNUM; PRINT.≠NUM= ≠,NUM;
DIGTUP=≠0123456789≠;
DIGITS=NL.;
(∀1<=N<=↓DIGTUP) (DIGTUP(N))IN, DIGITS;;
PRINT.≠DIGITS= ≠, DIGITS;
COMPUTE; FINISH;
```

<u>Output -- Program 91</u>

```
≠QNUM= ≠ ≠25≠
≠NUM= ≠ 25
≠DIGITS= ≠ ≤≠7≠ ≠4≠ ≠9≠ ≠1≠ ≠6≠ ≠3≠ ≠8≠ ≠0≠ ≠5≠ ≠2≠≥

* * * (END OF FILE ON INPUT) * * *
```

10.5. The FROM. Operator

The FROM. operator is used in the form:

A FROM. B;

where B is a set. When this statement is executed an arbitrary element of the set B is selected, removed from B and assigned as value to A. A FROM. B is therefore equivalent to

A = ARB. B; A OUT. B;

Program 92

```
LINE STATE
 NO   NO

          /* TO ILLUSTRATE A FROM. B */
          DO;
             B=≤2,5,8,12≥; PRINT.B;
          A FROM. B;
          PRINT.B; PRINT.A;
          A FROM. B;
          PRINT.B; PRINT.A;
          COMPUTE; FINISH;
```

Output -- Program 92

```
≤8 2 12 5≥
≤2 12 5≥
8
≤12 5≥
2

* * * (END OF FILE ON INPUT) * * *
```

10.6. The NOOP Instruction

A "no operation" statement is occasionally useful. For example, one may wish to use such a statement simply to have something to which a label may be attached. For this reason, SETLB provides a NOOP instruction which does nothing, as illustrated by the next program.

Program 93

```
LINE STATE
 NO   NO

          /* TO ILLUSTRATE THE NOOP */
          DO;
             T=<2,5,8,12>; PRINT,T;
             IF T(3)EQ,8 THEN NOOP,
             ELSE PRINT,≠THIS WILL NOT APPEAR≠;;
          COMPUTE; FINISH;
```

Output -- Program 93

```
<2 5 8 12>

* * * (END OF FILE ON INPUT) * * *
```

10.7. The IS., or General Assignment to the Right, Operator

In programming one often wishes to assign a name to a particular subexpression of a larger expression. SETLB allows this to be done by use of the IS. operator. A IS. B is a valid expression, having A as value, but also assigning the value of A to B. If A is not a variable but a total expression which we desire to assign as value to B, then the SETLB evaluation-order rules require that all of A be included in parentheses. If "A IS. B" is part of some larger expression, then "A IS. B" should itself be included in parentheses. All this is illustrated in the next program.

Program 94

```
LINE STATE
 NO    NO

            /* AN ILLUSTRATION OF THE IS. OPERATOR */
            DO;
               A=2; B=3; C=4; D=5;
            PRINT,(((((A*B) IS. X1)+ C  IS. X2)*D) IS. X3);
            PRINT,X1,X2,X3;
            COMPUTE; FINISH;
```

Output -- Program 94

```
50
6 4 50

* * * (END OF FILE ON INPUT) * * *
```

10.8. NEWAT.

On occasion, one requires a value which is simply different from every other value previously referenced from within a program. For this purpose SETLB provides the NEWAT. operator. It is guaranteed to give a value distinct from any other object or value that has been referenced in the entire program.

In the illustrative program below the value of NEWAT. is printed out five times in succession. Each time, a different value is returned.

Program 95

```
LINE STATE
 NO   NO

          /* ILLUSTRATION OF THE NEWAT. FUNCTION */
          DO;
             (∀1<=K<=5) PRINT. NEWAT.;
          COMPUTE; FINISH;
```

Output -- Program 95

```
BLK1
BLK2
BLK3
BLK4
BLK5

* * * (END OF FILE ON INPUT) * * *
```

10.9. Object Types

In order to test whether a particular entity is an integer, tuple, set, character string, bit string (any logical value) or a blank, one uses the SETLB TYPE. operator. In the program which follows, each of these is individually tested against the special constants INT., TUPL., SET., STR., BITS. and BLK. respectively. As usual, it is imperative to parenthesize appropriately when dealing with Boolean expressions. Apart from this the next program requires no explanation.

Program 96

```
LINE STATE
 NO   NO

          /* TYPE TESTER */
          DO;
             A=3; B=<4,5>; C=≤:1,3,5,7≥;
             D=≠HOHUM≠; E=T.; F=NEWAT.;
          PRINT.(TYPE.A) EQ. INT.;
          PRINT.(TYPE.A) EQ. TUPL.;
          PRINT.(TYPE.A) EQ. SET.;
          PRINT.(TYPE.A) EQ. STR.;
          PRINT.(TYPE.A) EQ. BITS.;
          PRINT.(TYPE.A) EQ. BLANK.;
          COMPUTE; DO;
          PRINT.(TYPE.B) EQ. INT.;
          PRINT.(TYPE.B) EQ. TUPL.;
          PRINT.(TYPE.B) EQ. SET.;
          PRINT.(TYPE.B) EQ. STR.;
          PRINT.(TYPE.B) EQ. BITS.;
          PRINT.(TYPE.B) EQ. BLANK.;
          COMPUTE; DO;
          PRINT.(TYPE.C) EQ. INT.;
          PRINT.(TYPE.C) EQ. TUPL.;
          PRINT.(TYPE.C) EQ. SET.;
          PRINT.(TYPE.C) EQ. STR.;
          PRINT.(TYPE.C) EQ. BITS.;
          PRINT.(TYPE.C) EQ. BLANK.;
          COMPUTE; DO;
          PRINT.(TYPE.D) EQ. INT.;
          PRINT.(TYPE.D) EQ. TUPL.;
          PRINT.(TYPE.D) EQ. SET.;
          PRINT.(TYPE.D) EQ. STR.;
          PRINT.(TYPE.D) EQ. BITS.;
          PRINT.(TYPE.D) EQ. BLANK.;
          COMPUTE; DO;
          PRINT.(TYPE.E) EQ. INT.;
          PRINT.(TYPE.E) EQ. TUPL.;
          PRINT.(TYPE.E) EQ. SET.;
          PRINT.(TYPE.E) EQ. STR.;
          PRINT.(TYPE.E) EQ. BITS.;
          PRINT.(TYPE.E) EQ. BLANK.;
```

Program 96 (Continued)

```
COMPUTE; DO;
PRINT.(TYPE.F) EQ. INT.;
PRINT.(TYPE.F) EQ. TUPL.;
PRINT.(TYPE.F) EQ. SET.;
PRINT.(TYPE.F) EQ. STR.;
PRINT.(TYPE.F) EQ. BITS.;
PRINT.(TYPE.F) EQ. BLANK.;
COMPUTE; FINISH;
```

Output -- Program 96

```
TRUE
FALSE
FALSE
FALSE
FALSE
FALSE

FALSE
TRUE
FALSE
FALSE
FALSE
FALSE

FALSE
FALSE
TRUE
FALSE
FALSE
FALSE

FALSE
FALSE
FALSE
TRUE
FALSE
FALSE

FALSE
FALSE
FALSE
FALSE
TRUE
FALSE

FALSE
FALSE
FALSE
FALSE
FALSE
TRUE


* * * (END OF FILE ON INPUT) * * *
```

10.10. The ASSERT Debug-Print Statement

The ASSERT statement is part of the SETLB debug package which provides facilities which will be discussed in more detail in a later section. ASSERT allows any Boolean (or, for that matter other) expression to be asserted to be true at any particular point in a SETLB program. This expression is evaluated. If its value is TRUE, nothing is printed. However, if its value is FALSE, the value of every variable occurring in the expression is printed; moreover, each such variable value is printed accompanied by the symbolic name of the variable. Thus the ASSERT statement provides a convenient way for printing variables, in a manner very convenient for debugging.

The following example illustrates the use of the ASSERT statement.

Program 97

```
LINE STATE
 NO   NO

        /* ASSERT = PRINT OUT VALUES IF EXPRESSION IS FALSE */
        DO; A=1; B=2; C=3;
        PRINT,(A+B)EQ,(C*A),≠THIS IS THE FIRST≠;
        ASSERT((A+B)EQ,(C*A));
        PRINT,(A*B)EQ,(C*A),≠THIS IS THE SECOND≠;
        ASSERT((A*B)EQ,(C*A));
        COMPUTE; FINISH;
```

Output -- Program 97

```
TRUE ≠THIS IS THE FIRST≠
FALSE ≠THIS IS THE SECOND≠
≠ASSERTION FAILED≠
- - - A 0 IN MAIN
      A IS 1
      B IS 2
      C IS 3

* * * (END OF FILE ON INPUT) * * *
```

10.11. Macros

Macros are symbols which represent whole passages of program text. Once a macro name has been associated with a body of text, each occurrence of the name is replaced by an occurrence of the body of text which it represents. Macro-names are associated with texts by using macrodefinitions.

The simplest form of a macrodefinition is:

+*<name of macro> = <body of text>**

The following example shows the use of some simple macros.

Program 98

```
LINE STATE
 NO    NO

/* USE OF A MACRO */
DO; +*MULL= COMPUTE;DO; **
+*ISH= COMPUTE; FINISH; **
A=≤:1,2,6≥; PRINT,A;MULL
B=≤:3,4≥; PRINT,B; ISH
```

Output -- Program 98

```
≤1 2 6≥

≤3 4≥

* * * (END OF FILE ON INPUT) * * *
```

Macros may also have parameters. The definition of a macro with parameters has the following general form

+*<macro name>(<parameter list>) = <body of text>**

An example would be

+*SAMPL(A,B) = IF 0 B A THEN Z ELSE Y**

Parameter text must be supplied whenever a macro with parameters is invoked. For example, the macro SAMPL which we have just defined might be invoked by writing

SAMPL(X,GT.) .

Given the preceding macrodefinition, this is precisely equivalent to an occurrence of the text:

IF 0 GT. X THEN Z ELSE Y .

On the other hand, the macro invocation SAMPL(S,→) is precisely equivalent to an occurrence of

IF 0 → S THEN Z ELSE Y .

Generally speaking, the parameter text supplied with a macro invocation is used to replace the occurrences of parameters in the body of text associated with the macrodefinition; then the resulting text is substituted for the invocation. The following example, which relates to the prime-generation code discussed in section 6.1.1, illustrates these general principles.

Program 99

```
LINE STATE
 NO   NO

/* A MACRO WITH PARAMETERS */
DO;
**ISPRIME(P)=(~2<=N<P↑(P//N) NE.0)**
PRINT.≤X,2<=X<=100↑ISPRIME(X)≥;
PRINT.[+:2<=Q<=100↑ISPRIME(Q)]Q;
COMPUTE; FINISH;
```

Output -- Program 99

```
≤97 17 2 83 67 3 19 5 37 53 71 7 23 89 73 41 11 43 59 61 13 29
 79 31 47≥
1060

* * * (END OF FILE ON INPUT) * * *
```

10.12. User-Defined Binary and Monadic Operators

SETLB allows the user to define not only functions and subroutines of ordinary form, but also binary infix operators and monadic prefix operators. The following preliminary example shows a function of ordinary form which determines which of two parameters is the larger. It is written in the ordinary way and is no different from any of the function subprograms we have encountered. We will soon show how this operator can be made available as a user-defined infix operator.

Program 100

```
LINE STATE
 NO    NO

            /* A FUNCTION IN THE REGULAR FORM */
            DO; DEFINEF BIGGER(A,B);
      BIGGER
            IF A GT.B THEN RETURN T.;
                       ELSE RETURN F.;;  END BIGGER;
            X=10; Y=5; PRINT,X,Y;
            IF BIGGER(X,Y) THEN PRINT,≠THE FIRST ARGUMENT IS BIGGER≠;
            ELSE PRINT,≠THE SECOND ARGUMENT IS BIGGER≠;
            END IF; COMPUTE; FINISH;
```

Output -- Program 100

```
10 5
≠THE FIRST ARGUMENT IS BIGGER≠

* * * (END OF FILE ON INPUT) * * *
```

By using a slightly different function header, we may define essentially the same function as a binary infix operator. This allows us to invoke the function by writing an expression with one of the arguments preceding the function name and the second following it. (It is for this reason that we speak of an <u>infix</u> operator.) Specifically, we invoke the function by writing

A BIGGER. B

This provides what is often a more natural form in which to use the function.

The following example shows the manner in which an infix operator is defined and used.

<u>Program 101</u>

```
LINE STATE
 NO   NO

        /* NOW IN THE FORM OF A BINARY FUNCTION */
        DO; DEFINEF A BIGGER.B;
        IF A GT. B THEN RETURN T.;
 BIGGERZZ
                    ELSE RETURN F.;;END A;COMPUTE;
        /* THIS SEPARATION OF BLOCKS IS ESSENTIAL FOR THIS PROGRAM TO WORK*/
        DO; X=7; Y=9; PRINT.X,Y;
        IF X BIGGER.Y THEN PRINT.≠THE FIRST ARGUMENT IS BIGGER≠;
                    ELSE PRINT.≠THE SECOND ARGUMENT IS BIGGER≠; END IF X;
        COMPUTE; FINISH;
```

<u>Output -- Program 101</u>

```
7 9
≠THE SECOND ARGUMENT IS BIGGER≠

* * * (END OF FILE ON INPUT) * * *
```

User-defined monadic prefix operators are similar to the binary infix operators except that they have a single argument which follows the name of the operator separated by a period. A monadic operator name, like a binary operator, is terminated by a period.

This feature is illustrated in the following program in which an "integer version" of the Newton-Raphson method of determining the square root of a number is used.

<u>Program 102</u>

```
LINE STATE
 NO    NO

         /* A PSEUDO-SQUARE ROOT ROUTINE AS A MONADIC FUNCTION */
         DO;
                 DEFINEF ISQRT. N;
                 IF N LE. 1 THEN RETURN N;;
  ISQRTZZ
                 A = N/2;               /* FIRST GUESS. */
                 (WHILE (A*A) GT. N) /* WHILE A IS TOO BIG: */
                   A = (A + N/A)/2;  /*  ITERATE. */
                 END;
                 RETURN A;              /* CAN YOU PROVE THAT THIS IS
                                           FLOOR(SQRT(N)) FOR N ≥ 2. */
                 END ISQRT.;
         COMPUTE; DO;
            (~1<=N<=100 ↑ ((N//5) EQ.1))
         PRINT.N,ISQRT,N; END ~1;
         COMPUTE; FINISH;
```

<u>Output -- Program 102</u>

```
1 1
6 2
11 3
16 4
21 4
26 5
31 5
36 6
41 6
46 6
51 7
56 7
61 7
66 8
71 8
76 8
81 9
86 9
91 9
96 9


* * * (END OF FILE ON INPUT) * * *
```

QUESTIONS

Chapter 10

1. Write a function subprogram which computes the absolute value of the difference between two numbers.
2. Assume two tuples of equal length with integer components. Write a program to print the larger of each pair of corresponding components.
3. Assume the tuple:

   T = <5,9,4,3,2,1>;

   Write a program to form a new tuple composed of components of the original tuple, ordered randomly.
4. Write a function subprogram which computes the absolute difference of the maximum value element of a set A of integers and the minimum value element of a set B of numbers.
5. Assume the following four sets:

   A = ≤: 5,7,9,<1,2>,'HI'≥

   B = ≤: 8,91,3, 'GO',<'BLUE'>≥

   C = ≤: 3,5,4,6,18,9,8≥

   D = ≤: 8,9112-1,8,11≥

   Write a subroutine which uses the operators IN. and OUT. to compute and print out the union of the first and third sets and the difference of the second and fourth sets.
6. Write a subroutine which uses the FROM. and MAX. operators to select an arbitrary element from sets A and B and prints out the maximum of the two.
7. Write a function subprogram using the FROM. operator to convert a set into a tuple.
8. Which of the following instructions, each of which uses the IS. operator, is valid:
   - (a) (A+B) IS. C);
   - (b) (A+B IS. C);
   - (c) ((A+B) IS. C) + 2 * C;

9. Define values for A and B. Write a program with the following effect: if A is less than B, C is equal to A times B; . Otherwise, C is equal to B^2.
10. Using the monadic operators HD. and TL., write a program to determine whether the head of a tuple is equal to the sum of the components of the tail.
11. Show that adding NEWAT. to a set three times in succession increases the number of elements in that set by three.
12. Write a subroutine which takes each element of a set S of numbers and prints it out together with a statement indicating whether it is odd or even.
13. Assume a set S:

```
S = ≤: <4,3>, ≤: 5,6,8≥ , 8, 'YUM' ≥;
```

Write a program which selects and prints each element from the set S together with a statement of whether it is a set, tuple, integer or a character string.

14. Define a generalized function RAND which behaves in the following fashion:
    (a) if the argument is an integer, it generates a random integer, 1 to the integer.
    (b) if the argument is a string, a random character of that string is returned.
    (c) if the argument is a tuple, the function will select a random component of the tuple.
    (d) if the argument is a logical value (bit string), a random logical value is generated.
    (e) if the argument is a set, it will select a random element from the set (not necessarily the same as ARB.).
    (f) if the argument is anything else, it is merely returned.
15. What will be the printed output of the following program:

```
DO;
   A=3; B=2; C=1;
PRINT.(A+B) EQ. C;
ASSERT(A+B) EQ. C);
COMPUTE: FINISH;
```

16. The following program uses three macros. What output if any, will it generate?

```
DO;
    +*BLK=COMPUTE; DO; **
    +*LAST=COMPUTE; FINISH;**
    +*T(A,B)=(TUP(A))(B)**
    TUP=<<1,2,3>,<4,5,6>,<7,8,9>>;
    BLK
    PRINT. T(3,3), T(2,1);
    LAST
```

17. Define a binary operator which returns the first N characters of a given string.
18. Define a monadic operator which eliminates the vowels from a given character string.
19. Define a monadic operator which sorts the components of a tuple of character strings into alphabetical order.
20. Define a monadic operator which, given a set of tuples of character strings, produces a set whose elements are the same tuples but arranged in alphabetical order.

11. READING FROM DATA CARDS

None of the programs discussed and illustrated so far have read data from an external medium; in each case, the program contained all the information to perform a desired calculation. However, it is sometimes essential to read data from cards during the execution of a program. This may be done quite simply, in a manner which we shall now explain.

(1) The following two control cards are inserted immediately after the ID card and before the ATTACH, SETLABS, SETLB. control card.

```
COPYBR(INPUT,INFILE)
REWIND(INFILE)
```

(2) The SETLB program must begin with:

```
DO; INFILE= MAKFILE('INFILE',72); COMPUTE;
```

(3) Data must be punched in a "stream" form, which we shall now explain.

a) Sets are punched in the manner exemplified by:

≤1 2 3≥

Note that in punching a set on a data card we use no colon, no commas and no semicolon.

b) Tuples are punched in the manner shown by the following example.

<5 6 7 8>

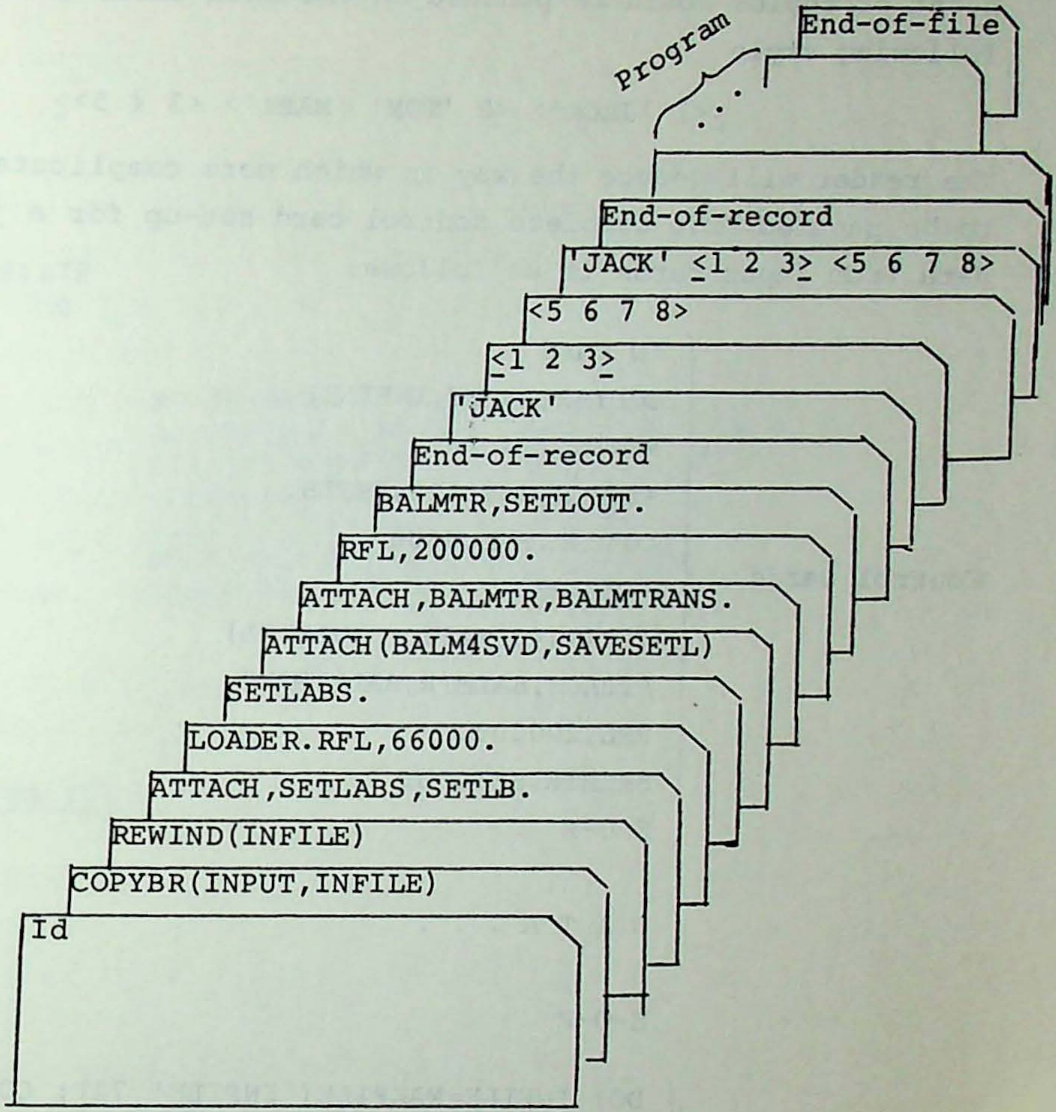
Note that we use no commas between the components.

c) Character strings are punched in the following style:

'JACK'

with the quote sign fore and aft.

(4) The data to be read is placed after the control cards and separated from them by an end-of-record card. After the data comes another end-of-record card followed by the SETLB program. At the end of the prgram, as always, is the end-of-file card. Thus the whole deck structure appears as shown in the following sketch:


Program
End-of-file
. . .
End-of-record
'JACK' <1 2 3> <5 6 7 8>
<5 6 7 8>
<1 2 3>
'JACK'
End-of-record
BALMTR,SETLOUT.
RFL,200000.
ATTACH,BALMTR,BALMTRANS.
ATTACH(BALM4SVD,SAVESETL)
SETLABS.
LOADER.RFL,66000.
ATTACH,SETLABS,SETLB.
REWIND(INFILE)
COPYBR(INPUT,INFILE)
Id

A set of tuples would be punched on the data cards in the following style

$\underline{<}$<1 'JACK'> <2 'TOM' 'MARY'> <3 4 5>$\underline{>}$.

The reader will deduce the way in which more complicated sets are to be punched. The complete control card set-up for a job reading data from input cards is as follows:

```
                 ID card
                 COPYBR(INPUT,INFILE)
                 REWIND(INFILE)
                 ATTACH,SETLABS,SETLB.
                 LOADER.RFL,66000.
Control Cards    SETLABS.
                 ATTACH(BLM4SVD,SAVESETL)
                 ATTACH,BALMTR,BALMTRANS.
                 RFL,200000.
                 BALMTR,SETLOUT.
                 E-O-R

                 D A T A . . .

                 E-O-R

                 DO; INFILE=MAKFILE('INFILE',72); COMPUTE;
   Program       ....
                 FINISH;
                 E-O-F
```

The following program illustrates the SETLB input-output conventions:

<u>Program 103</u>

```
LINE STATE
 NO   NO

          /* TO READ DATA FROM CARDS */
          DO;INFILE=MAKFILE(≠INFILE≠,72);COMPUTE;
          DO;READ,NAME;PRINT,NAME;COMPUTE;
          DO;READ,SET;PRINT,SET;COMPUTE;
          DO;READ,TUPLE;PRINT,TUPLE;COMPUTE;
          DO;READ,NAME,SET,TUPLE;COMPUTE;
          DO;PRINT,NAME,SET,TUPLE;COMPUTE;FINISH;
```

<u>Output -- Program 103</u>

```
≠JACK≠

≤1 2 3≥

<5 6 7 8>

≠JACK≠ ≤1 2 3≥ <5 6 7 8>

* * * (END OF FILE ON INPUT) * * *
```

12. SOME SAMPLE PROGRAMS.

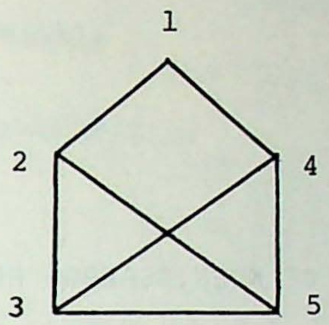
12.1 A First Full-Scale Example of the Use of SETLB: The Koenigsberg Bridge Problem

We now present a more substantial program as an indication of the ease with which combinatorial problems can be attacked in SETL. The problem to be discussed goes back to Euler (1707-83). He considered a question whose generalization is now known as the Koenigsberg Bridge problem, which arose from the pattern in which the various parts of Koenigsberg city were connected by bridges. Presumably like many citizens out for a stroll and reluctant to walk back along their path, he sought to discover whether it was possible to start out from some place in the city and from there to cross each bridge in the city only once.

This problem, which Euler saw could be formulated as a problem concerning abstract graphs, is discussed at length in "Graphs and Their Uses" by Oystein Ore of Yale University, published by the L. W. Singer Company. It is in fact the problem of traversing all the edges of a given graph once and only once. Its fame derives from the fact that Euler's work on this problem initiated the subject of topology, which has by now grown to be one of the most beautiful and important parts of mathematics.

Euler discovered that all the edges of a graph can be traversed once and only once if and only if the graph contains either 0 or 2 points at which an odd number of edges come together. If it contains no such points, then one can start one's traversal anywhere. If it contains 2 such points, one must start at one of these points, and finish at the other. Euler's rule for traversal is as follows: as successive edges are traversed, they are removed from the graph. At each moment, one attempts to traverse an edge leading to a point which can still be reached along some alternate path. If this is impossible, it follows that only one edge can possibly be traversed; this is traversed (and, of course, removed) and the process continues.

These rules are illustrated by the following graph:



In the above graph 3 edges come together at each of the 4 points 2,3,4,5. Hence the graph cannot be traversed without repeating any edge. This graph is the first one submitted to the program given below. Given this graph, the program prints "impossible graph".

Next, a horizontal arc connecting node 2 to node 4 is added to the graph. Then only the two points 3,5 have an odd number of incoming edges. This makes traversal possible as the output, showing the path 3,2,1,4,2,5,3,4,5 as the path of traversal, indicates.

In the program below graphs are represented by sets whose elements are themselves 2-element sets. Each 2-element set represents a pair of points connected by an edge. For example, the graph depicted above is represented by the set

S = ≤:≤:1,2≥,≤:2,3≥,≤:2,5≥,≤:1,4≥,≤:4,5≥,≤:4,3≥,≤:3,5≥≥ .

To add the edge 2,4 to the grpah, one simply executes

S = S WITH. ≤:2,4≥;

The SETLB program which follows is profusely annotated, and the reader should be able to follow its logic.

Program 104

```
LINE STATE
 NO   NO

          /*    EULERS -BRIDGES OF KOENIGSBERG- PROGRAM                              */
          DO;
                                        /* THIS PACKAGE OF ROUTINES ENCODES
                                           L.EULERS ALGORITHM FOR THE -BRIDGES OF*/
                                        /* KOENIGSBERG- PROBLEM. THE PROBLEM IS:
                                           GIVEN A GRAPH CONSISTING OF NODES AND */
                                        /* EDGES, TO TRACE IT OUT, TRAVERSING
                                           EVERY EDGE (BRIDGE) ONCE AND ONLY ONCE,
                                           (NODES MAY BE VISITED MORE THAN ONCE) */
                                        /* EULERS ALGORITHM IS AS FOLLOWS:
                                              A. THE PROBLEM CAN BE SOLVED ONLY IF
                                           THE NUMBER OF NODES AT WHICH AN ODD
                                           NUMBER OF EDGES COME TOGETHER IS
                                           EITHER 0 OR 2.                           */
                                        /*    B. IF THERE IS A NODE AT WHICH AN ODD
                                           NUMBER OF EDGES COME TOGETHER, START
                                           AT SUCH A NODE. OTHERWISE, START
                                           ANYWHERE.                                */
                                        /*    C. IF YOU ARE AT A NODE P WHICH
                                           HAS A NEIGHBOR Q WHICH CAN BE REACHED */
                                        /* FROM P BY AN INDIRECT PATH NOT USING THE
                                           EDGE<P,Q>, STEP TO Q. IF NOT, P WILL  */
                                        /* HAVE ONLY ONE NEIGHBOR Q. STEP TO THAT,
                                              D. IN STEPPING FROM P TO Q, ERASE  */
                                        /* THE EDGE <P,Q> OF THE GRAPH.             */
                                        /* REPEAT STEPS (D) AND (E) UNTIL EVERY
                                           EDGE HAS BEEN TRAVERSED.                 */
                                        /*    ***IF YOU CAN, PROVE THAT THE
                                           ALGORITHM WORKS..,  ***                  */
                                        /*                                          */
                                        /* THIS MACRO ALLOWS EASY ADDITION OF AN
                                           EXTRA ELEMENT TO A TUPLE                 */
                **ENDOF(T)=T((#T)+1)**
                DEFINEF WALK(GR);
      WALK
                LOCAL NODES,X,P,ODDS,Q,NP,GRAPH;
                                        /* THIS IS EULERS ALGORITHM. IT ASSUMES
                                           THAT THE GRAPH IS GIVEN AS A SET OF
```

Program 104 Continued

```
                                   2-ELEMENT SETS ≤:P,Q≥; THE SET ≤:P,Q≥
                                   REPRESENTS AN EDGE CONNECTING P AND Q,*/
          GRAPH=COPY(GR);
                                /* BEGIN BY FORMING THE SET OF ALL NODES
                                   IN THE GRAPH                            */
          NODES=[+:X→GRAPH] X;
                                /* NEXT, CONVERT THE GRAPH FROM A SET OF
                                   UNORDERED PAIRS ≤:P,Q≥ TO A SET OF
                                   ORDERED PAIRS <P,Q>, ALWAYS INCLUDING */
                                /* BOTH <P,Q> AND <Q,P> IN THE NEW VERSION
                                   OF THE GRAPH. THIS MAKES IT POSSIBLE
                                   TO USE THE SETL -FUNCTIONAL MAPPING-
                                   OPERATIONS IN REMAINDER OF THE CODE     */
          GRAPH=(TFSET[GRAPH] IS, Q)+R[Q];
                                /* THE -TFSET- FUNCTION IS DEFINED BELOW */
                                /* FORM THE SET OF NODES WHICH AN ODD
                                   NUMBER OF EDGES COME TOGETHER,          */
                                /* IF THIS HAS MORE THAN TWO ELEMENTS,
                                   GRAPH CANT BE TRACED                    */
          IF(+(≤P→NODES↑((+GRAPH≤P≥)//2) NE,0≥
                           IS, ODDS)) GT, 2 THEN
          PRINT,≠IMPOSSIBLE GRAPH≠; RETURN OM,; END IF;
                                /* CHOOSE A STARTING POINT                 */
          P=IF ODDS NE, NL, THEN ARB,ODDS ELSE ARB,NODES;
                                /* START PATH                              */
          PATH=<P>;
          (WHILE GRAPH NE, NL, DOING
                                /* REMOVE FROM GRAPH THE EDGE TRAVERSED
                                   ON EACH CYCLE                           */
                 (<P,Q>) OUT, GRAPH;
                 (<Q,P>) OUT, GRAPH;
                 ENDOF(PATH)=Q; P=Q;)
                                /* APPLY EULERS RULE FOR CHOOSING NEXT
                                   POINT OF PATH                           */
          IF ∃ Q→(GRAPH≤P≥ IS, NP)↑Q→TRANC(GRAPH LESS,<P,Q>,P)
               THEN NOOP;
               ELSE Q=ARB,NP; END IF;
          END WHILE;
          RETURN PATH;
          END WALK;
          DEFINEF TRANC(GRAPH,X);
TRANC
          LOCAL SET, NEW;
                                /* THIS IS A -TRANSITIVE CLOSURE- ROUTINE.
                                   GIVEN A GRAPH REPRESENTED BY A SET OF
                                   ORDERED PAIRS, IT FORMS THE SET OF ALL*/
                                /* POINTS WHICH CAN BE REACHED FROM THE
                                   POINT X BY A PATH IN THE GRAPH          */
          SET=GRAPH≤X≥; NEW=SET;
          (WHILE NEW NE, NL,)
               NEW=GRAPH[NEW]-SET;
               SET=SET+NEW;
          END WHILE;
          RETURN SET;
          END TRANC;
                                /*                                         */
                                /* WE NOW GIVE TWO SMALL AUXILIARY
                                   ROUTINES. THE FIRST, TFSET, CONVERTS
                                   A TWO ELEMENT SET ≤:P,Q≥ INTO AN
                                   ORDERED PAIR <P,Q>;                     */
```

Program 104 (continued)

```
              DEFINEF TFSET(X); LOCAL Y,Z;
   TFSET
              Y FROM, X; Z FROM, X;
              RETURN<Y,Z>;
              END TFSET;
                                /* THE NEXT ROUTINE JUST REVERSES AN
                                   ORDERED PAIR                              */
              DEFINEF R(X); RETURN <X(2),X(1)>; END R;
     R
                                /* HERE IS A SAMPLE GRAPH, IT CAN¬T BE
                                   TRACED, SINCE IT HAS FOUR NODES AT WHICH
                                   AN ODD NUMBER OF EDGES COME TOGETHER.     */
         GRAPH=≤;≤:1,2≥,≤:1,4≥,≤:2,3≥,
               ≤:2,5≥,≤:4,3≥,≤:4,5≥,≤:3,5≥≥;
         PRINT,GRAPH;
         PRINT,WALK(GRAPH);
                                /* BY ADDING AN EXTRA ARC, WE MAKE
                                   IT POSSIBLE TO TRACE THE GRAPH.           */
         GRAPH=GRAPH+≤:≤:2,4≥≥;
         PRINT,GRAPH;
         PRINT,WALK(GRAPH);
         COMPUTE; FINISH;
```

Output -- Program 104

```
≤≤4 5≥ ≤2 3≥ ≤1 2≥ ≤1 4≥ ≤3 4≥ ≤2 5≥ ≤3 5≥≥
*IMPOSSIBLE GRAPH*
OM.
≤≤4 5≥ ≤2 3≥ ≤1 2≥ ≤1 4≥ ≤3 4≥ ≤2 5≥ ≤2 4≥ ≤3 5≥≥
<3 2 1 4 2 5 3 4 5>

* * * (END OF FILE ON INPUT) * * *
```

12.2. A Second Full-Scale Example of the Use of SETLB: Translating to Pig Latin.

The following program consists of three subroutines. The first, a programmed monadic operator called TILB., breaks off the first part of any string supplied to it, using the first blank character as a dividing mark. It returns a pair consisting of the substring preceding and the substring following the blank. If there is no blank, a pair consisting of the whole input string and of a null character string is returned. In examining this routine, note the manner in which a SETLB existential is used to test for the existence of a blank character, and to locate the first blank character if it exists, all at once.

The second subroutine is written as a binary operator and is called TRAN. . The first argument to the operator is a dictionary DICT; the second argument is a string of words, separated by blanks, to be translated using the dictionary. The routine TRAN. uses TILB. repeatedly, to break off one word after another from its input string. Each word WD broken off is 'looked up' in the dictionary, by evaluating DICT(WD). If it is 'found in the dictionary', i.e., if DICT(WD) is not OM., then DICT(WD) is appended to a translation string built up progressively by TRAN.; if DICT(WD) is OM., corresponding to the case in which WD is not present in the dictionary, then WD itself, followed by the parenthesized remark 'NOT IN DICTIONARY', is inserted into the developing translation. In examining the routine TRAN., note the manner in which a WHILE iterator is used to control the statements which build the translated string; the iteration terminates when the input INP has been reduced to the null character string. Note also the manner in which the IS. operator is used twice, once to save the pair returned by TILB. as the value of the variable PR , and then immediately again to save the first component, i.e. HD., of this pair as the value of the variable WD.

The first part of the output is produced by using a miniature 4-word "English to German dictionary". Once this set of pairs is supplied, we have the DICT('THE') = 'DAS', etc., permitting 'translation'. Of course, a simple table look-up procedure like the one programmed here is quite incapable of translating natural language with anything like the sophistication required in less trivial and artificial cases.

Our next step is to replace the set DICT., which is used in the procedure TRAN., by a programmed function PIGD. Note here the important fact which this illustrates: that SETLB variables are free to take on values of changing type, and that the expression DICT(WD) can be evaluated whether DICT denotes a set of tuples or a programmed function. As distinct from the 4-word tabular dictionary used first, PIGD can translate any English word into Piglatin. In accordance with the rules of Piglatin, it does this by moving the initial consonants of the word to its end, and adding 'AY'. However, if the word begins with a vowel, it is prefixed with 'P' and 'AY' is affixed.

In examining the routine PIGD, note the use of substring and concatenate operations, and note also the use of an existential to locate the first vowel in a word.

The above remarks, together with the comments contained in the SETLB program which now follows, should make the program readable.

The program's output includes a partial trace initiated by the (HELP) control card option. This trace gives a fairly detailed 'motion' picture of the program's action. Line 5 contains instructions whose net effect is to turn on the trace for TRANS only. For further details see section 14.9.

Program 105

```
STATE
 NO

  0      /* A TABLE LOOKUP -TRANSLATOR- ROUTINE WITH AN AUXILIARY   */
  0      /* STRING BREAK-UP ROUTINE; ALSO, A 4 - WORD -ENGLISH TO   */
  0      /* GERMAN- DICTIONARY, FINALLY, A PROGRAMMED PIGLATIN      */
  0      /*                  DICTIONARY                             */
  0      NOCHECK STORES; CHECK STORES(TRANS);
  0      DO;
  0         DEFINEF TILB,STR; LOCAL X;
TILBZZ
  0                                    /* ROUTINE TO BREAK OFF THE FIRST WORD */
  0                                    /* OF AN INPUT STRING USING LEFTMOST   */
  0                                    /* BLANK TO DETERMINE BOUNDARY OF FIRST*/
  0                                    /* WORD
  0
  0         IF Ǝ1<=N<=+STR+STR(N) EQ,≠ ≠ THEN
  1            X=STR(1:N);
  2         ELSE
  2            X=STR;  STR=NULC.; RETURN<X,NULC.>;
  4         END IF;
  4         IF N EQ, + STR THEN STR = NULC.; ELSE STR=STR(N+1:(+STR)-N).
  7         RETURN<X(1:(+X)-1),STR>;
  7         END TILB.;
  0
  0         COMPUTE; DO;
  0         DEFINEF DICT TRAN,INP; LOCAL TRANS,DW,X;
TRANZZ
  0         TRANS=NULC.;
  1         (WHILE((HD,((TILB,INP)IS,FR))IS,WD)NE,NULC.)
  2            INP=FR(2);
  3            IF(DICT(WD) IS, X) NE, OM, THEN DW=X;
```

Program 105 (Continued)

```
5            ELSE DW=WD+≠(NOT IN DICTIONARY)≠;
6            END IF;
6            TRANS=TRANS+DW+≠ ≠;
7        END WHILE;
7        RETURN TRANS;
7        END DICT TRAN.;
0
0    DICT=≤:<≠THE≠,≠DAS≠>,<≠WATER≠,≠WASSER≠>,<≠IS≠,≠IST≠>,<≠COLD≠,≠KALT≠>≥;
1        COMPUTE; DO;
1        TEXT=≠THE WATER IS VERY COLD WATER≠; PRINT,TEXT;
3        PRINT,DICT TRAN,TEXT;
4    COMPUTE; DO;
4
4        VOWS=≤:≠A≠,≠E≠,≠I≠,≠O≠,≠U≠≥;
5
5        DEFINEF PIGD(WD); LOCAL N,K;
   PIGD
0            IF∃1<=N<=↓WD↑WD(N)∈VOWS THEN
1               IF N EQ. 1 THEN RETURN ≠P≠+WD+≠AY≠;
2               ELSE /* N GT. 1 */ K=N-1; RETURN WD(N:(↓WD)-K)+WD(1:K)+≠AY≠;
3               END IF N;
3        ELSE /* ALL CONSTANTS */ RETURN WD;
3        END IF;
3           END PIGD;
5        COMPUTE; DO;
5
5        TEXT2=≠NOW IS THE TIME FOR ALL GOOD MEN TO LEARN SETL≠;
6        PRINT,TEXT2;
7        PRINT,PIGD TRAN,TEXT2;
8    COMPUTE; FINISH;

********** SUSPICIOUS VARIABLES (USED LESS THAN  3 TIMES.**********
PR          VOWS
********************************

 +++  SETLB MACROS VERSION 2.4    MAY 2, 1973.   +++
COMMENT ≠PHASE 2 OUTPUT FROM SETLB TO BALMSETL TRANS≠
  COMMENT ≠COMPILATION TERMINATED NORMALLY ----≠
TIMES (SEC), TOTAL=  5.86, PASS1=  1.38, PASS2=  4.45

= = = END SETLB = = =
```

Output -- Program 105 (continued)

```
≠THE WATER IS VERY COLD WATER≠
- - - ENTERING TRANZZ
- - - AT 1 IN TRANZZ TRANS IS ≠≠
- - - ENTERING TILBZZ
- - - RETURN FROM TILBZZ AT 7 WITH VALUE <≠THE≠ ≠WATER IS VERY COLD WAT
 ER≠>
- - - AT 7 IN TRANZZ TRANS IS ≠DAS ≠
- - - ENTERING TILBZZ
- - - RETURN FROM TILBZZ AT 7 WITH VALUE <≠WATER≠ ≠IS VERY COLD WATER≠>

- - - AT 7 IN TRANZZ TRANS IS ≠DAS WASSER ≠
- - - ENTERING TILBZZ
- - - RETURN FROM TILBZZ AT 7 WITH VALUE <≠IS≠ ≠VERY COLD WATER≠>
- - - AT 7 IN TRANZZ TRANS IS ≠DAS WASSER IST ≠
- - - ENTERING TILBZZ
- - - RETURN FROM TILBZZ AT 7 WITH VALUE <≠VERY≠ ≠COLD WATER≠>
- - - AT 7 IN TRANZZ TRANS IS ≠DAS WASSER IST VERY(NOT IN DICTIONARY) ≠

- - - ENTERING TILBZZ
- - - RETURN FROM TILBZZ AT 7 WITH VALUE <≠COLD≠ ≠WATER≠>
- - - AT 7 IN TRANZZ TRANS IS ≠DAS WASSER IST VERY(NOT IN DICTIONARY) K
ALT ≠
- - - ENTERING TILBZZ
- - - RETURN FROM TILBZZ AT 4 WITH VALUE <≠WATER≠ ≠≠>
- - - AT 7 IN TRANZZ TRANS IS ≠DAS WASSER IST VERY(NOT IN DICTIONARY) K
ALT WASSER ≠
- - - ENTERING TILBZZ
- - - RETURN FROM TILBZZ AT 4 WITH VALUE <≠≠ ≠≠>
- - - RETURN FROM TRANZZ AT 7 WITH VALUE ≠DAS WASSER IST VERY(NOT IN DI
CTIONARY) KALT WASSER ≠
≠DAS WASSER IST VERY(NOT IN DICTIONARY) KALT WASSER ≠

≠NOW IS THE TIME FOR ALL GOOD MEN TO LEARN SETL≠
- - - ENTERING TRANZZ
- - - AT 1 IN TRANZZ TRANS IS ≠≠
- - - ENTERING TILBZZ
- - - RETURN FROM TILBZZ AT 7 WITH VALUE <≠NOW≠ ≠IS THE TIME FOR ALL GO
 OD MEN TO LEARN SETL≠>
- - - ENTERING PIGD
- - - RETURN FROM PIGD AT 3 WITH VALUE ≠OWNAY≠
- - - AT 7 IN TRANZZ TRANS IS ≠OWNAY ≠
- - - ENTERING TILBZZ
- - - RETURN FROM TILBZZ AT 7 WITH VALUE <≠IS≠ ≠THE TIME FOR ALL GOOD M
 EN TO LEARN SETL≠>
- - - ENTERING PIGD
```

Output -- Program 105 (continued)

```
- - - RETURN FROM PIGD AT 2 WITH VALUE ≠PISAY≠
- - - AT 7 IN TRANZZ TRANS IS ≠OWNAY PISAY ≠
- - - ENTERING TILBZZ
- - - RETURN FROM TILBZZ AT 7 WITH VALUE <≠THE≠ ≠TIME FOR ALL GOOD MEN
 TO LEARN SETL≠>
- - - ENTERING PIGD
- - - RETURN FROM PIGD AT 3 WITH VALUE ≠ETHAY≠
- - - AT 7 IN TRANZZ TRANS IS ≠OWNAY PISAY ETHAY ≠
- - - ENTERING TILBZZ
- - - RETURN FROM TILBZZ AT 7 WITH VALUE <≠TIME≠ ≠FOR ALL GOOD MEN TO L
 EARN SETL≠>
- - - ENTERING PIGD
- - - RETURN FROM PIGD AT 3 WITH VALUE ≠IMETAY≠
- - - AT 7 IN TRANZZ TRANS IS ≠OWNAY PISAY ETHAY IMETAY ≠
- - - ENTERING TILBZZ
- - - RETURN FROM TILBZZ AT 7 WITH VALUE <≠FOR≠ ≠ALL GOOD MEN TO LEARN
 SETL≠>
- - - ENTERING PIGD
- - - RETURN FROM PIGD AT 3 WITH VALUE ≠ORFAY≠
- - - AT 7 IN TRANZZ TRANS IS ≠OWNAY PISAY ETHAY IMETAY ORFAY ≠
- - - ENTERING TILBZZ
- - - RETURN FROM TILBZZ AT 7 WITH VALUE <≠ALL≠ ≠GOOD MEN TO LEARN SETL
 ≠>
- - - ENTERING PIGD
- - - RETURN FROM PIGD AT 2 WITH VALUE ≠PALLAY≠
- - - AT 7 IN TRANZZ TRANS IS ≠OWNAY PISAY ETHAY IMETAY ORFAY PALLAY ≠
- - - ENTERING TILBZZ
- - - RETURN FROM TILBZZ AT 7 WITH VALUE <≠GOOD≠ ≠MEN TO LEARN SETL≠>
- - - ENTERING PIGD
- - - RETURN FROM PIGD AT 3 WITH VALUE ≠OODGAY≠
- - - AT 7 IN TRANZZ TRANS IS ≠OWNAY PISAY ETHAY IMETAY ORFAY PALLAY OO
DGAY ≠
- - - ENTERING TILBZZ
- - - RETURN FROM TILBZZ AT 7 WITH VALUE <≠MEN≠ ≠TO LEARN SETL≠>
- - - ENTERING PIGD
- - - RETURN FROM PIGD AT 3 WITH VALUE ≠ENMAY≠
- - - AT 7 IN TRANZZ TRANS IS ≠OWNAY PISAY ETHAY IMETAY ORFAY PALLAY OO
DGAY ENMAY ≠
- - - ENTERING TILBZZ
- - - RETURN FROM TILBZZ AT 7 WITH VALUE <≠TO≠ ≠LEARN SETL≠>
- - - ENTERING PIGD
- - - RETURN FROM PIGD AT 3 WITH VALUE ≠OTAY≠
- - - AT 7 IN TRANZZ TRANS IS ≠OWNAY PISAY ETHAY IMETAY ORFAY PALLAY OO
```

Output -- Program 105 (continued)

```
DGAY ENMAY OTAY ≠
- - - ENTERING TILBZZ
- - - RETURN FROM TILBZZ AT 7 WITH VALUE <≠LEARN≠ ≠SETL≠>
- - - ENTERING PIGD
- - - RETURN FROM PIGD AT 3 WITH VALUE ≠EARNLAY≠
- - - AT 7 IN TRANZZ TRANS IS ≠OWNAY PISAY ETHAY IMETAY ORFAY PALLAY OO
DGAY ENMAY OTAY EARNLAY ≠
- - - ENTERING TILBZZ
- - - RETURN FROM TILBZZ AT 4 WITH VALUE <≠SETL≠ ≠≠>
- - - ENTERING PIGD
- - - RETURN FROM PIGD AT 3 WITH VALUE ≠ETLSAY≠
- - - AT 7 IN TRANZZ TRANS IS ≠OWNAY PISAY ETHAY IMETAY ORFAY PALLAY OO
DGAY ENMAY OTAY EARNLAY ETLSAY ≠
- - - ENTERING TILBZZ
- - - RETURN FROM TILBZZ AT 4 WITH VALUE <≠≠ ≠≠>
- - - RETURN FROM TRANZZ AT 7 WITH VALUE ≠OWNAY PISAY ETHAY IMETAY ORFA
Y PALLAY OODGAY ENMAY OTAY EARNLAY ETLSAY ≠

≠OWNAY PISAY ETHAY IMETAY ORFAY PALLAY OODGAY ENMAY OTAY EARNLAY ETLSAY
 ≠


* * * (END OF FILE ON INPUT) * * *
```

13. SUMMARY OF SETLB FEATURES

The following pages summarize most of the SETLB language features. Not all of these features are described in this introductory text. Those features which are described are accompanied by page references.

Feature		SETLB Representation	References in this text
1.	Objects		
1.1	Atoms	Examples	
	Integer	0, 2, 567, -9	57
	Character string	≠ABC≠	122
	Label	LABEL: X=Y	114
	Boolean	1B	
	Blank	Created by function NEWAT.	147
	Undefined blank	OM.	41,48
	Subroutine	DEFINE	130
	Function	DEFINEF	130
	Current limitations: variable names should not exceed 8 characters and period-delimited operator names should not exceed 6 characters.		
1.2	Sets	$\underline{<}:X_1,X_2,\ldots,X_n\underline{>}$	11
1.3	Tuples	$<X_1,X_2,\ldots,X_n>$	39
1.4	Type function	TYPE.	148
	Types	INT., SET., TUPL., STR., LAB., BITS., BLANK., SUBR.	
1.5	Special constants:		
	Null set	NL.	11
	Null-string	NULC.	122
	Null-tuple	NULT.	50
	True	T.	14,17
	False	F.	14,17
	Undefined	OM.	41,48

	Feature	SETLB Representation	References in this Text
2.	Operations		
2.0	No-operation	NOOP;	145
2.1	Arithmetic operators:		
	Plus	+	57
	Minus	-	57
	Times	*	57
	Divide	/	57
	Residue	//	56
	Maximum	MAX.	139
	Minimum	MIN.	139
	Absolute value	ABS.	138
2.2	Comparison and Boolean Operators:		
	Equals	EQ.	20
	Not equal	NE.	20
	Less than	LT.	20
	Less-equal	LE.	20
	Greater than	GT.	20
	Greater-equal	GE.	20
	Includes	INCS.	18
	And	A. or AND.	20
	Or	O. or OR.	20
	Not	N. or NOT.	20
2.3	Character string:		
	Decimal convert	DEC.	143
	Octal convert	OCT.	
	Catenate	+	122
	Repeat	*	
	Substring	C(I:J)	125
	Length	↓	122

	Feature	SETLB Representation	References in this text
2.4	Set operations:		
	Membership	X→A	16
	Number	↓	29
	With	WITH.	31,33,68,142
	Less	LESS.	31,33,142
	Lesf	LESF.	
	Diminish	X OUT. S	142
	Augment	X IN. S	142
	Diminish-f	X OUTF. S	
	Diminish and retrieve	X FROM. S	144
	Union	+	27
	Intersect	*	27
	Difference	-	27
	Symmetric difference	//	28
	Arbitrary element	ARB. S	34
	Powerset	POW(S)	25
	N-element subsets	NPOW(N,S)	25
2.5	Tuple operations:		
	Head	HD.	50
	Tail	TL.	50
	Component	T(I)	
	Length	↓	39
	Catenation	+	40
2.6	General set former:		
	Form SET where:		
	B(X) is an expression in X; X→A; X is an element of A	≤E(X), X→A≥	11
	Alternative: N ranges over integers X to get	≤E(N), 1<=N<=M≥	95
	only those E(X) such that condition C(X) is true	≤E(X), X→A↑C(X)≥	
	Complex i form	≤E(X,N),X→A,1<=N<=M↑C(X,N)≥	

	Feature	SETLB Representation	References in This Text
2.7	Set applications		
	Application	$F(X)$ or $F(X_1,\ldots,X_N)$	73
	Multivalued application	$F\underline{<}X\underline{>}$ or $F\underline{<}X_1,\ldots,X_N\underline{>}$	80
	Range	$F[X]$ or $F[X_1,\ldots,X_N]$	79
2.8	Compound operators		
	[operator:iterator] expression		
	e.g.	[+: X→S]F(X)	90
		[*:1<N<=M,Y→S(N)↑C(N,X)]F(X,N)	93
2.9	Conditional expressions		
	IF condition THEN expression ELSE expression,		
	e.g., X = IF A GT. B THEN A+B ELSE A*B;		117
	more generally IF condition THEN expression ELSE IF condition THEN expression ... ELSE expression		
2.10	General assignment to the right operation	IS.	146
3.	Control and Iteration Statements		86
3.1	Unconditional GO TO LABEL (can only be used in function or subroutine body)		114
3.2	Conditional statements:		
	IF condition THEN block;		108
	or IF condition THEN block ELSE block;		109
	or IF condition_1 THEN block_1 ELSE IF condition_2 THEN block_2 ... ELSE block_n;		110

Conditional statements can be ended with a semicolon; or with the terminator 'END;'; or with 'END IF;' etc.

Feature	SETLB Representation	References in this Text

3.3 Iteration statements

a) Iterate over a block as long as a condition is fulfilled

(WHILE condition) block; 110

(WHILE condition DOING block_1) block_2; 113

WHILE scopes can be terminated with a semicolon; or with the terminators 'END;' or 'END WHILE;' , etc.

b) Iteration over elements of sets

For all X in S, repeat block 95

(∀ X → S) block;

For all K in (M,N) repeat block 105

(∀ M <= K <= N) block;

Iterations with decreasing iteration index, etc. are also available.

c) Compound iterators

(∀ X → S, M(X) <= K <= N(X) ↑ C(X,Y)) block; 107

Iteration scopes, like WHILE scopes, can be terminated in several ways.

3.4 Quantified Boolean expressions

a) Existential quantifiers

Exists X in S such that C(X): ∃ X → S ↑ C(X) 59

Exists K in (M,N) such that C(X): ∃ M <= K <= N ↑ C(X) 64

Note: If the value of such an expression is true, then the variable X or K is set equal to the first element in the range for which C(X) is true.
Existentials which search in decreasing order of iteration index, and various compound existentials, are also available.

b) Universial quantifiers

For all X in S such that C(X): ∀ X → S ↑ C(X) 62

For all K in (M,N) such that C(X): ∀ M <= K <= N ↑ C(X) . 64

Feature	SETLB Representation	References in this Text

Note: If the value of such an expression is false, then the variable X or K is set equal to the first element in the range for which C(X) is false.

Various compound universals such as

	V X → A ↑ ≡ Y → B ↑ X EQ. Y	66

are also available.

4. Functions, Subroutines and Operator Definitions

4.1 Functions:

Feature	SETLB Representation	References in this Text
Function	DEFINEF FNC(ARG1,...,ARGK)	131
Monadic form of operator	DEFINEF MON. ARG	
Binary form of operator	DEFINEF P1 BIN. P2	
Return value from within function	RETURN EXPRESSION	

4.2 Subroutines:

Feature	SETLB Representation	References in this Text
Subroutine	DEFINE SUB(ARG1,...,ARGK)	
Monadic form	DEFINE MON. ARG	138
Binary form	DEFINE P1 BIN. P2	139
Return	RETURN	130

4.3 Macros:

Feature	SETLB Representation	References in this Text
Without arguments	+* G = COMPUTE **	151
or	+* HDT = HD. TL.**	
With arguments	+*TWOSET(A,B,C)=A=HD. C, B=HDT C**	152

Macro definitions within macros, etc., are available.

5. Sinister Forms

Name = expression;

HD. NAME = expression;

TL. NAME = expression;

NAME$(X_1,\ldots,X_n)$ = expression;

NAME$\le X_1,\ldots,X_k \ge$ = expression;

NAME$(X_1,\ldots,X_k)$ = OM.;

NAME(N1:N2) = expression; (substring assignment)

Feature	SETLB Representation	References in this Text
6. Input - Output		
	PRINT. $EXPN_1,\ldots,EXPN_k$;	4
	READ. $NAME_1,\ldots,NAME_k$;	163
	WRITE. $NAME_1,\ldots,NAME_k$;	

Input/output is file-oriented.

The following four files are available for I/O.

a) INPUT - system input file from which the source text is read by the compiler

b) INFILE - file from which data can be read by means of a READ statement. This file is the file INPUT by default.

c) OUTPUT - system output file on which the output from the compiler is written. Data can be written on this file by PRINT. statement.

d) OUTFILE- file on which data can be written by means of the WRITE. statement. It is equal to the file OUTPUT by default.

Redefinition of the files INFILE and OUTFILE can be achieved by statements such as:

```
OUTFILE = MAKFILE(LOCALFILENAME, LINEWIDTH);
```

or

```
INFILE  = MAKFILE(LOCALFILENAME, LINEWIDTH);
```

where

LOCALFILENAME is a character string, and
LINEWIDTH is an integer.

E.g.

```
OUTFILE = MAKFILE('MYOUT', 120)
```

will cause all output produced by subsequent WRITE. statements to be routed to the system (SCOPE) file named MYOUT, and grouped into 120 character lines.

Feature	SETLB Representation	References in this Text
7. Miscellaneous		
Compile-block opener:	DO;	4
Incremental compilation command:	COMPUTE;	4
Required terminator for complete SETLB program:	FINISH;	4
Debug statement which prints value of all variables in expression, if expression has any value other than TRUE	ASSERT<expression>	150

8. Additional Information Concerning Iterator Scopes

A scope is opened by either:

1. a 'FORALL' iterator	86
2. a 'WHILE' iterator	110
3. a 'DO;' statement	4
4. a subroutine or function definition.	130

Each such scope must be closed by a corresponding END-element, which may have one of the following forms (form C below is the preferred one since it allows scopes to be verified at compile time):

a) an extra semicolon

b) END;

c) END followed by up to 4 tokens other than semicolon followed by a semicolon.

Examples:

c1) (∀ X → S) X = X+1; END ∀X;

c2) (WHILE X → S DOING X = X+1;) Y=X; END WHILE X→S;

c3) DEFINE A OP. B; ...; END A OP.;

Example c1 shows END followed by 2 tokens.

Example c2 shows END followed by 4 tokens.

14. MISCELLANEOUS ADVANCED INFORMATION

1. SETLB Operator Precedence

Only two operator precedence levels are used

a) Various Boolean-valued comparison operators bind more strongly than other operators. These include:

EQ. NE. GT. GE. LT. LE. INCS.

b) Other operators associate to the right. E.g.

A-B-C	means	A-(B-C)	etc.

Please beware of the effects of this rule.

c) Monadic operators have minimal scope

-A+B	means	(-A)+B

and

N.X A.Y	means	(N.X) A. Y

But note that N. A EQ. B means N. (A EQ. B) ,

and

↓A EQ. B	means	↓(A EQ. B) .

2. Syntactic Precedence levels

Four syntactic precedence levels exist. These are in order of increasing tightness of binding:

expression, factor, element and atom.

a) Expressions are made up of factors. Factors are separated by operators.

b) The first factor of an expression is the scope of any monadic operator (or sequence of monadic operators prefixed to the expression). Existential and universal quantifiers, and compound operators are treated much like monadic operators.

c) An atom is

i) either a 'special quantity name' (e.g. NEWAT.,T.,NULC.)

ii) or a lexical constant (e.g. an integer or a string)

iii) or a parenthesized or bracketed subexpression

iv) or a name functionally applied to a sequence of argument subexpressions, e.g.,

$$F\underline{<}arg_1,\ldots,arg_n\underline{>}, F(arg_1,\ldots,arg_n), F[arg_1,arg_n]$$

d) An element is either

i) an atom

ii) or an atom functionally applied as in (iv) above.

3. Code Blocks within Expressions

SETLB 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 used in this way is prefixed by a [: , terminated by a] and should contain at least one statement of the form

RETN expression;

The value of the expression in the first such statement executed defines the value of the entire block. Example
Example:

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

4. Local and Global Variables in Subroutines and Functions

As has been indicated previously, SETLB is a preprocessor to BALMSETL (which is an extension of BALM obeying all the syntactic and semantic conventions of BALM). In particular the name-scoping rules are those of BALM. This fact has the following consequences: A variable not declared explicitly is global. To declare variables to be local to a given subroutine or function, use a LOCAL statement. Such a statement has the following form:

LOCAL NAMEA, NAMEB, ... ;

It must appear as the first statement within the procedure body.

This statement causes the variables in the list following the keyword "LOCAL" to be local to the subroutine or function within which it appears.

5. Modification of Variable Values by Subroutines

BALM is address and pointer-oriented in its treatment of compound objects. SETL, on the other hand, is intended to be a consistent 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. The indicated logical discrepancy between SETL and BALM raises a number of problems of which it is important to be aware. One such class of problems concerns subroutine arguments. In BALM, subroutines may generally not modify their arguments since BALM restores all subroutine arguments to their pre-call values immediately on return from a call. However, 'simple" objects such as integers, truth values, etc., are directly represented in BALM by their values, while "compound" objects such as tuples and sets are represented in BALM by a pointer to the memory address at which an array containing its elements are stored. Modifications of a compound object leave the pointer to the object unchanged, but change the body of the object. It follows that after a subroutine call, changes in compound objects occurring as arguments will be propagated back to a calling routine. Changes in simple objects will not be propagated back in the same way. To avert possible difficulties arising from this fact it may be necessary to use a variable known globally both to a called and to a calling routine in order to get an effect normally obtained by using a subroutine which modifies its arguments. Alternately one can make use of a function returning a vector value, followed by a multiple assignment, i.e. one can rewrite a subroutine.

```
    DEFINE SUB(X,Y,Z); ... X= ...; Y= ...; Z = ...; RETURN;
```

as a function

```
    DEFINEF SUB(X,Y,Z); ...X= ...; Y= ...; Z = ...;
    RETURN <X,Y,Z>;
```

and call the subprocedure in the form

<A,B,C> = SUB(A,B,C);

The following example involving a subroutine and a main program, illustrates some of the difficulties discussed above.

Program 106

```
LINE STATE
 NO    NO

        /* ANOMALIES IN ALTERING DATA OBJECTS AND RETURNING  */
            /* PARAMETERS FROM SUBROUTINES */
        /* THESE ANOMALIES ARE EXPLAINED BY THE FACT THAT:   */
        /*    A, THE -BALM- LANGUAGE UNDERLYING SETLB IS A   */
        /*       POINTER LANGUAGE, NOT PERMITTING PERMANENT  */
        /*       SUBROUTINE MODIFICATION OF ARGUMENTS        */
        /*    B, -COMPOUND- STRUCTURES IN BALM ARE           */
        /*       REPRESENTED BY POINTERS TO THEIR BODIES     */
        DO; DEFINE G(U,V,W,X,Y,Z);
      G
          U=T,;V=NL,;X=W;
          (3) IN. X;
          (2) IN. W;
          Y=Y+1;
          Z(2)=1;
          RETURN;
          END G;
          U=F,; V=OM,; W=≤:1≥; X=≤:1≥; Y=1; Z=<3,2>;
          G(U,V,W,X,Y,Z);
        PRINT.U,V,W,X,Y,Z;
        COMPUTE; FINISH;
```

Output -- Program 106

```
FALSE OM. ≤1 2 3≥ ≤1≥ 1 <3 1>

* * * (END OF FILE ON INPUT) * * *
```

6. BALMSETL Reserved Words

a) The following is an alphabetical list of the reserved words of the BALMSETL system which underlies the SETLB system. These identifiers should not be used as names of user created variables in SETLB. The letter following each word has the following significance:

U: Unary operator
I: Infix operator
B: Bracket operator
P: Procedures
V: Other global variables
M: Macro keywords
P*: BALM primitive which acts like a procedure

ABS	U	INEG	P*	READ	P
AND	I	INFIX	P	RDLINE	P*
APPEND	P	INR	M	REMARK	P*
ARB	U	INTERSCT	P	REMINFIX	P
ARGUMENT	P*	INTQ	P*	REMMACRO	P
ATOM	U	IS	M	REMUNARY	P
AUGMENT	P	LAND	P*	REPEAT	I
BACKSPAC	P*	LBLQ	P*	RESTAT	P
BE	M	LE	I	RESUMEAL	P*
BEGIN	B	LENGTH	P	RETURN	U
BLANK	V	LESF	I	REWIND	P*
BLANKQ	P	LESFN	I	RPLACA	P*
BCF	I	LESS	I	RPLACD	P*
BCFN	I	LET	M	SAVEALL	P*
BRACKET	P	LEVEL	V	SAVEBALM	P
BREAKUP	P	LFROMV	P	SAVESETL	P
BSTRQ	P	LIST	P*	SAVSTAT	P
CFRCMV	P*	LOR	P*	SETINDEX	P
CLOSE	P*	LOGQ	P*	SETMODE	P*
CODEQ	P*	LOOKUP	P	SETOF	M
COMP	I	LT	I	SETPROPL	P*
COMPILE	P	MACRO	P	SETPROPY	P
COMPL	P*	MAKFILE	P	SETQQ	P
CONCAT	P	MAKPROPS	P*	SETSUB	P*
CONCATV	P*	MACVAR	V	SETSUBV	P*
CONSTRUC	P	MAKALOCA	P	SETVALUE	P*
COPY	P	MAKVECTO	P*	SFRMOID	P*
CRASHMAX	V	MAKVLOCA	P	SFRMOV	P*
DEC	U	MAPX	P	SHIFT	P*
DFD	U	MAX	I	SIM	I
DIMINISH	P	MEMBER	P	SIZ	P
DIMF	P	MIN	I	SIZE	U
DIMFN	P	MODE	P*	SKIPWD	P

DO	B	NE	I	SOF	I
DSLSH	I	NELT	U	SOFN	I
DUMMY	P	NEXTELT	P	STKTRACE	P*
DUP	P	NEQUAL	I	STRING	P*
EL	I	NEWAT	P	STRINGOF	U
ELSE	I	NIL	V	STOP*	P
ELSEIF	I	NILQ	P*	SUB	P*
END	I	NILVECT	V	SUBST	P
ENDFILE	P*	NL	V	SUBV	P*
EQ	I	NOMEGAP	P	SUCH	M
EQUAL	I	NOT	U	SYMDIF	P
ERROR	P	NPOW	P	SYSLIST	V
EXECUTE	P	NULB	V	TAIL	U
EXISTS	M	NULC	V	TAILN	I
EXPAND	P	NULL	U	TAILSPEC	U
FALSE	V	NULLSET	V	TALKATIV	V
FIRSTWD	P	NULT	V	THEN	I
FOR	U	NULLTUPL	V	TIME	P*
FORALL	M	NUMARGS	P*	TL	U
FR	M	OCT	U	TRACE	V
FROMSET	P	OCTMODE	V	TRANSLAT	P
GARBCOLL	P*	OF	I	TRUE	V
GE	I	OFN	I	TTYFLAG	V
GENSET	P	OMEGAP	P	TUPQ	P
GENSYM	P	OPEN	P*	TYPE	U
GENTUP	P	OR	I	UNARY	P
GETPROP	P	ORDINAL	P	UNDEF	P
GETWD	P	PAIRQ	P*	UNDFD	U
GO	U	PAIRTUP*	P	UNION	I
GOTO	U	PL	U	VALUE	U
GT	I	POW	P	VECTOR	P*
HD	U	PRINT	P	VFROML	P
HEAD	U	PROC	B	VFROMS	P
IDENTQ	P	PROCTRAC	P	WHERE	M
IDFROMC	P*	PROPL	P*	WHILE	U
IDFROMS	P*	PROTECT	P*	WITH	I
IDQ	P*	PRT	P*	WRLINE	P*
IF	U	PRTMAP	P	XOR	P*
IFROMID	P	PTRACE	P	ZR	U
IN	M	QUOTE	P*		
INCS	I	QUANT	N		
INDEX	P	RANDOM	P		

b) SETLB Use of BALMSETL Functions

Many of the BALMSETL reserved words listed above refer to functions of interest and may be used in SETLB. For example:

STRINGOF(OBJ)	This function returns a character string which is the external representation of the object OBJ .

c) SETLB-BALMSETL Correspondences

It is useful --especially for debugging purposes-- to know the correspondence between the symbols used to represent various SETLB features and the BALMSETL keywords into which these are translated. Most of these names remain the same after translation or are only slightly changed and can be easily recognized when they are printed out as part of the SETLB debugging information. Those whose translated forms are not so obvious are listed here.

SETLB	BALMSETL
SUBR.	CODEQ
OM.	UNDEF
//	MOD
C(I:J)	SUB(C,I,J)
↓ (when used for length of a string)	SIZE
X → A	X EL A
↓ (when used for #elements in a set)	NELT
X OUT. S	DIMF(S,X)
X IN. S	AUGMENT(S,X)
X OUT. S	DIMF(S,X)
≤:X1,...,XN≥	GENSET(X1,...,XN)
F(X)	F OF X
F≤X≥ F	F SOF X
F[X]	F BOF X

7. <u>Additional Name Restrictions</u>

a) Variable names should not exceed 8 characters and period-deliminated operator names should not exceed 6 characters.

b) Names ending in ZZ are specially used by the preprocessor to translate period-delimited operator names and should generally be avoided.

8. <u>Control Card Parameters</u>

The SETLB translator provides several options which the user may select by supplying a list of the necessary keywords on the control card initiating execution of SETLB. The keywords and their interpretation are as follows:

a) XRF, the cross-reference option. If this option is selected the output file will include a complete cross reference map for all names in the SETLB input program.

b) LIST, code list option. If this option is selected the output file from the translator will include a listing of the BALMSETL code generated.

c) ABT, error-abort option. If this option is selected, the translator will abort if any errors are detected in the SETLB source, or if an internal translator error occurs (such as overflow of a needed array) which would make production of BALMSETL meaningless. If the compiler aborts, a dayfile message will indicate the reason. If the compiler terminates normally, the dayfile message ≠SETLB DONE≠ is issued

d) SYSERR, system-error-trace option. If this option is selected, the output listing will contain additional information which may help trace errors in the translator. The additional output is quite voluminous and has an obscure form; thus this option should be used only by someone familiar with the SETLB translator and one who is trying to track down the source of a potential translator error.

e) SL, initial value of SETLISTC control word. Default is 01; note that SETLISTC word consists of six bits, which are interpreted as follows (counting from left to right, one to six).

   1 - restore saved word
   2 - save current word, set word to value given
   3 - list tokens sent to parser
   4 - not used
   5 - list macro definitions
   6 - list input cards as read in

   Note that SL value on control card is octal, e.g. SL= 13.

f) SLO, SETLISTC over-ride. Default is off. If enabled, occurrences of SETLISTC within input text are ignored; so initial value of SETLISTC word holds for entire run.

g) L, label option, which provides for up to a ten-character label for the run (label must not contain commas (,)). Note that the first statement of the BALMSETL prelude is

   JOBLABEL = USERLABL;

   where USERLABEL is user-supplied label. This provides for convenient labelling of BALMSETL input and of any save files produced.

h) BULL bulletin option. Default is on. If option is enabled, then the permanent file SETLBBULLETIN is copied to the user's output file. This is to provide users with immediate notice of system changes and problems.

i) HELP request debugging aids. Default is off. If debugging aids are requested, SETLB translator will insert calls to trace routines in the BALM system at key points of the user's source code. Details of the use of these features are described under the heading "debugging features" on page 194.

Use of Translator Options

Translator options are specified by providing a list of keywords and values, enclosed in parentheses, on the control card initiating execution of the translator. A keyword is assigned a value by following the keyword with an equals sign (=) and the value. The value must be a nonnegative integer, or one of the words ON, YES, T, or TRUE (which corresponds to a value of 1), or one of the words OFF, NO, F or FALSE (which corresponds to a value of 0). The default settings are as follows:

XRF = FALSE	i.e.	no cross-reference map produced
LIST= FALSE	i.e.	BALMSETL code will not be listed on output file
ABT = TRUE	i.e.	compiler will abort if errors occur
SYSERR= FALSE	i.e.	no optional system error-trace output provided

In this connection, note that all of the following control cards are equivalent:

```
SETLB.
SETLB. (XRF=FALSE, LIST=NO, ABT=TRUE, SYSERR=0)
SETLB. (L I S T = O F F)      (since blanks are ignored)
```

Here are some additional examples of parameter lists:

To obtain cross-reference map and list BALMSETL code on listing file, use

SETLB. (XRF,LIST)

To obtain no cross-reference map and no list of BALMSETL code use

SETLB.

To provide some trace information if an error occurred on a previous run and to force the compiler to produce BALMSETL even in the presence of errors, use

SETLB. (ABT=NO, SYSERR=YES)

9. Debugging Features

The SETLB translator provides very useful debugging features. At the user's request, the translator will insert calls to trace routines which the BALM system converts into the BALM code which it produces. At execution time the values of associated global variables may be set by the user to control the generation of debug output by the BALM trace routines. The trace features currently available provide for the tracing of program flow, entry and return to subprograms, and trace of assignments to selected variables.

As an example of how the trace package works, consider the SETLB sequence for the last few lines in a hypothetical procedure P:

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

With no debugging aids invoked, this translates into the BALM sequence:

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

If flow tracing is requested, then the BALM code produced is:

```
ATSN(2,=P),
A = 10,
ATSN(3,=P)
RETURN(A),
```

If, in addition, ENTRY/EXIT tracing is requested, the code generated is:

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

If stores to A are being traced, then the code is

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

Note that the form ≠=P≠ will print in BALM as -P- since the binary use of ≠ = ≠ is interpreted as the ≠quote≠ operator by the BALM system.

When the code shown above is executed, the output will be as follows:

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

If, however, the code is set up for output which is only to include the traces of assignments 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
```

The example illustrates three levels of user control of the debug options. The user may separately decide

A. Whether to generate calls to BALM trace routines;

B. Which kinds of calls to generate; moreover

C. The user has available execution time control over the debug output-nonoutput decision. This control is exercised by changing the values of global variables examined by the trace routines.

We shall now discuss these options in more detail.

9.1 Activating the Debug Package in SETLB Translator

a) Debug control card parameters for the SETLB translator are as follows:

ATSN	compile calls tracing program flow
ATEQ	compile calls tracing assignments
ATEX	compile calls tracing ENTRY and EXIT for routines
ATSNTRC	set initial value of trace switch ATSNTRC
ATEQTRC	set initial value of trace switch ATEQTRC
ATEXTRC	set initial value of trace switch ATEXTRC

If none of these options is selected, the output of the SETLB translator will not include debugging statements.

The special parameter for HELP is equivalent to the combination ATEX, ATSN, ATEQ, and is the easiest way of involving the most commonly used debug options.

b) SETLB Statements Controlling Debug Features

A "CHECK" statement is available in the SETLB language. This statement has the form:

<CHECK / NOCHECK> <FLOW / STORES / ENTRY> <NAMELIST>:

where '/' indicates that one of the options is allowed. The <NAMELIST> is optional, and if present, consists of a list of names, separated by commas, and enclosed in parentheses. A NAMELIST is only available for the STORES option. The initial situation assumed by the translator is:

```
CHECK FLOW;
CHECK STORES;
CHECK ENTRY;
```

Accordingly, the debug system's global trace-control switches are initialized as

```
ATSNTRC=F.;
ATEQTRC=T.;
ATEXTRC=T.;
```

With these settings effective at execution time, only assignments and ENTRY/EXIT traces will be printed.

With the above setting of its parameters, the translator will insert calls to ATSN at the start of each executable statement, calls to ATENTRY at start of each procedure, calls to ATEXIT before a RETURN statement, and calls to ATSET after assignment.

The user can control the insertion of the trace calls by inserting trace statements in his program. For example, to trace assignments to all variables use CHECK STORES; to trace assignments to all variables except X and Y, use CHECK STORES; NOCHECK STORES (X,Y); .

One of the results of using the (HELP) debugging feature is that the printed listing includes, in addition to a card sequence number under the heading LINE NO, another sequence number under the heading STATE NO, standing for "statement number". This latter number refers to the number of preceding executable clauses.

As stated above the BALMSETL system will initialize the trace print switches to be TRUE, except for flow trace switch ATSNTPC. The SETLB statement

```
ATSNTRC=T.;
```

should be executed to initiate full program flow trace.

As an additional example of the use of the trace switches, suppose that an error occurs just after the tenth pass through a loop on the variable I. By inserting the statement:

```
IF I GT. 10 THEN ATSNTRC = T.;;
```

in the loop, a full flow trace will be initiated only when I is greater than 10.

15. BEWARES

In this section, we list common SETLB pitfalls which may trouble the SETLB user and which he will have to learn to avoid. Most of these difficulties relate to the limitations of the current implementation and will not permanently be characteristics of the SETL language.

"Beware"

1. Right associativity
2. Local and global variables in procedures
3. Variable modification in procedures
4. Side-effects of compound structure
5. Reserved words
6. Name restrictions
7. Lables and go-to's within iterations
8. Subroutine arguments used as global variables
9. Effect of multiple assignments.
10. Distinction between IN. and → , OUT. and FROM.
11. Distinction between infix subroutine calls and infix operators
12. Errors for which no diagnostic or a confusing (perhaps, BALM-level) diagnostic is issued
13. If a set defines a function, then in order to change the value of the function for a particular argument one may use a sinister call, e.g., after executing

    $$S = \underline{<}<1,3>,<2,4>\underline{>};$$

    $$S(1) = 5;$$

    the new value of S will be

    $$S = \underline{<}<1,5>,<2,4>\underline{>} \quad .$$

    If S(1) is now used in an expression it will return the value 5. Please beware that in the current implementation only the simplest forms of sinister calls (such as the one shown above) will work.
14. The residue (remainder) operator cannot be used as a compound operator.
15. Whenever it is legal the use of parentheses is strongly encouraged.

Related Literature

Falkoff, A. D. and Iverson, K. E. APL/360 User's Manual. IBM Corporation, Thomas J. Watson Research Center, Yorktown Heights, New York (1968).

Morris, James B., A Comparison of MADCAP and SETL. University of California, Los Alamos Scientific Laboratory, Los Alamos, New Mexico (1973).

Schwartz, J. T. Abstract Algorithms and a Set Theoretic Language for Their Expression (preliminary draft), Computer Science Department, Courant Institute of Mathematical Sciences, New York University (1971).

Schwartz, J. T. On Programming: An Interim Report on the SETL Project, Installment I: Generalities. Computer Science Department, Courant Institute of Mathematical Sciences, New York University (1973).

Wells, M. B. Elements of Combinatorial Computing, Pergamon Press, Oxford (1971).

Index

Absolute value operator ABS. 138
Arbitrary value operator ARB. 34
Arithmetic in SETLB 56-57
ASSERT. statement 150

BALM 2,114
Binary operator 140,154
Boolean expressions 14,18,22,148
Boolean operators 20,22
Built-in functions 138

Changing sets 31
Character set 3
Character strings 122
Checking a formula 94
Comment card symbols 8-9
Comparison operators 20
Components 39
Compound operators 90,93
Conditional expressions 117
Continuation of cards 9
Control cards 5
Control statements 86
Crash 40

Data cards 160
Debugging 4,194
DEC. operator 143
Delimit sets 11,54
Dice 141
Difference of sets 27
Domain 73
Duplicate elements 11,13,30

Element 11,23
ELSE 108-109
Enumeration operator 29,122
Error termination 42,68
Existential quantifiers 59-61,107,127
-- locating an element 61
Explicit set formers 11

Finite domain 73
-- sets 73
Flowcharts 86
FROM. operator 144
Function 73
-- defined 120

GO TO 114
GO TO -less programming 116

Head operator (HD.) 50
HELP 6-7
IF 108-109
IN. operator 142
Including set operator (INCS.) 18
Indexing of tuples 41
Infinite sets 73
Integer arithmetic 6-7
Integer numbers only 57
Intersection of sets 29
IS. operator 146
Iterator 86

Koenigsberg Bridge problem 164

Index
(Continued)

Labels 114,145

LESS. operator 31,33,142

Lexicographic comparison 135

Macros 151

Maximum value operator (MAX.) 139

Member 13

Minimum value operator (MIN.) 139

-- as a compound operator 140

Modifying tuples 44,47

Monadic operators 154

Multiple assignments 67

Multivalued functions 73,80

NEWAT. operator 147

Newton-Raphson method 156

NOOP. instruction 145

NOT. 14

NPOW-n-element subset 25

Null character string(NULC.) 122

Null set (NL.) 11

Null tuple (NULT.) 50

Number of elements in a set 29

Object types 148

OM. 41,48

Ordered pairs 73

OUT. operator 142

Pig Latin problem 169

Power set (POW) 25

Procedure rules 22

Prime number generator 92

PRINT. 4

Printing fixed text 8

Random function 141

Range of a function 79

READ 163

Remainder (residue) operator 56

Scope 87

Selection operator 34

Set formers 52,57,107

Set equality 19

Set inclusion (INCS.) 18

Sets 11

Sorting 127,134

Spaces 13

Subprograms 130

Subroutines 130,132

Subset 13,18,19,23,52

Substrings 125

Such that 52

Summary of SETLB features 176

Symmetric difference 52

Tail of a tuple 44

TL. operator 50

THEN 108-109

Translator 169

Tuples 39

-- as a function 73

TYPE. operator 148

Undefined value 48

Union of sets 27

Universal quantifier 62,107

-- locating an element 63

WHILE 110

WITH. operator 31,33,68,142