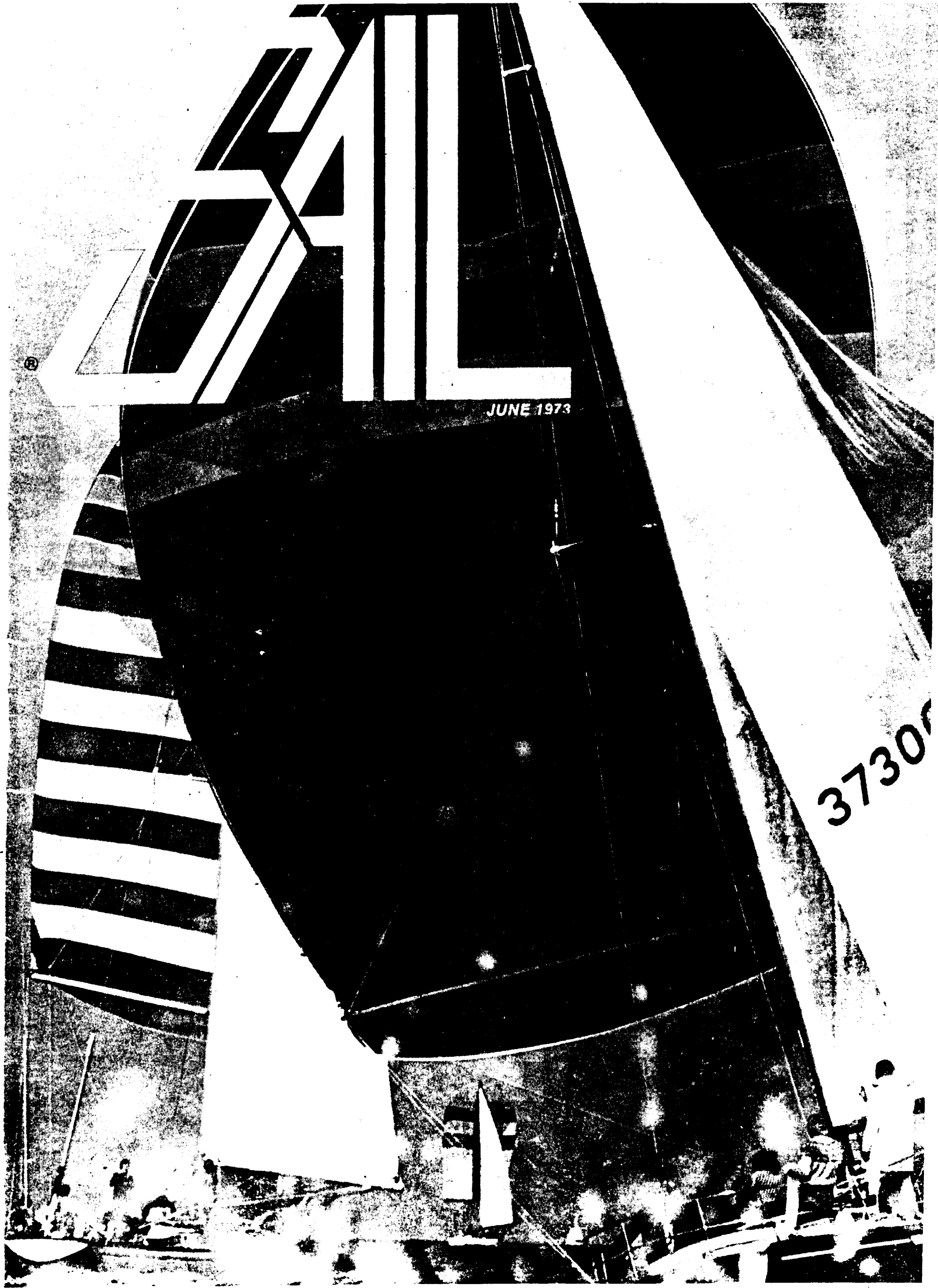

SAIL
®
JUNE 1973
3730

STANFORD ARTIFICIAL INTELLIGENCE LABORATORY **JULY 1973**
MEMO AIM-204

COMPUTER SCIENCE DEPARTMENT
REPORT STAN-CS-73-373

SAIL USER MANUAL

edited by

Kurt A. VanLehn

ABSTRACT

SAIL is a high-level programming language for the PDP-10 computer. It includes an extended ALGOL 60 compiler and a companion set of execution-time routines. In addition to ALGOL, the language features: (1) flexible linking to hand-coded machine language algorithms, (2) complete access to the PDP-10 I/O facilities, (3) a complete system of compile-time arithmetic and logic as well as a flexible macro system, (4) user modifiable error handling, (5) backtracking, and (6) interrupt facilities. Furthermore, a subset of the SAIL language, called LEAP, provides facilities for (1) sets and lists, (2) an associative data structure, (3) independent processes, and (4) procedure variables. The LEAP subset of SAIL is an extension of the LEAP language, which was designed by J. Feldman and P. Rovner, and implemented on Lincoln Laboratory's TX-2 (see [Feldman & Rovner]). The extensions to LEAP are partially described in "Recent Developments in SAIL" (see [Feldman]).

This manual describes the SAIL language and the execution-time routines for the typical SAIL user: a non-novice programmer with some knowledge of ALGOL. It lies somewhere between being a tutorial and a reference manual.

This manual was supported by the Advanced Research Projects Agency of the Office of the Secretary of Defense under Contract No. SD183 (order number 457), National Institute of Mental Health Contract No. PHS MH 06645-12, and National Science Foundation Contract No. GJ-776.

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of any of the funding agencies.

We would like to thank Bernard A. Goldhirsh and the Institute For the Advancement of Sailing for their kind permission to use the cover design of the June 1973 issue of SAIL magazine.

Reproduced in the USA. Available from the National Technical Information Service, Springfield, Virginia 22151.

PREFACE

HISTORY OF THE LANGUAGE

The GOGOL III compiler, developed principally by Dan Swinehart at the Stanford Artificial Intelligence Project, was the basis for the non-LEAP portions of SAIL. Robert Sproull joined Swinehart in incorporating the features of the LEAP language, developed by J. Feldman and P. Rovner on the Lincoln Laboratory's TX-2, into SAIL. The first version of the language was released in November, 1969. Since then, the language has been maintained, expanded, and improved by many people. Foremost amoung these are Russell Taylor, Jim Low, and Hanan Samet. They were responsible for the introduction into the language of processes, procedure variables, interrupts, contexts, matching procedures, the new macro system, and many other features.

USING THIS MANUAL

For the first reading, a light skim of sections 1 through 4 followed by a careful perusal of subsection 19.1 should be adequate to familiarize the new user with the differences between ALGOL and SAIL and allow him to start writing programs in SAIL. The other sections of this manual are relatively self contained, and can be read when one wants to know about the features they describe. The exceptions to this rule are sections 10, 11, and 12. These describe the basics of the LEAP and are essential for understanding of the following sections. Much of the implementation information contained in older versions of this manual has been moved to the appendices and a forthcoming implementation manual.

An attempt has been made to keep forward references to a minimum. In other words, if the manual is freely using concepts unfamiliar to you, they are probably defined in an earlier section. However, the definitions of some common concepts such as "variable" and "identifier" have been left until section 19.

CHANGES IN THE LANGUAGE

One of the design goals for the current Implementation of SAIL was to retain, as far as possible, compatibility with previous versions. We have been fairly successful in retaining source language compatibility, but not completely successful, since other design considerations frequently proved to be overriding. Most of these exceptions occur with constructs that, while never explicitly illegal, were never quite "legal" either. Essentially, this means that programs which contain "hacks" may or may not be able to run unchanged For Instance, assignment of an integer to the datum of a set item will cause horrible things to happen when the item is deleted. One should consult the appropriate sections of this manual, for detailed information. Other notable incompatibilities include:

1. The procedure implementation was somewhat changed. This change should not adversely affect any programs that do not use START-CODE or link to assembly language routines. However, for efficiency the user may want to consider declaring some of his smaller procedures SIMPLE. The new implementation required that another register ('12) be dedicated to SAIL's exclusive use. Programs that modify this register do so at their utmost peril.

2. Non-own sets are deallocated when the block in which they are declared is exited.

3. The storage management system for arrays has been modified. Again, this change can only affect programs that allocate arrays using START-CODE.

WARNING: This list is primarily intended as a general guide to the most outstanding incompatibilities, and should not be construed as being complete. Users are strongly urged to read over the manual, since doing so will introduce them to the new features of the language, some of which are quite useful, as well as informing them of any subtle changes in the old semantics. In any event the experience at Stanford was that conversion of programs proved to be surprisingly easy. The only real holdouts were a couple of giants that made heavy and subtle use of START-CODE blocks and assembly language routines.

UNIMPLEMENTED CONSTRUCTS

The following items are described in the manual as if they existed. As the manual goes to press, they are not implemented. The are listed in the probable order of their implementation.

1. NEW (<context-variable>). Creates a new item which has a datum that is a context.

2. Using a <context-variable> instead of a list of variables in any of the REMEMBER, FORGET or RESTORE statements.

3. Using ∞ in the expression n of REMOVE n FROM list.

4. ANY⊗ANY≡ANY searches in Leap. That is, any search where no constraints at all are made on the triple returned.

5. CHECKED itemvars. The dynamic comparison of the datum type of an item to the datum type of the CHECKED itemvar that the item is being assigned to. Currently, for example, if you assign an item with an integer datum to an itemvar that was declared a string itemvar, no check is performed. It is the user's responsibility to see

that the datum is not subsequently not accessed, for if it is, it will be treated as a string,

TABLE OF CONTENTS

SECTION PAGE

1 PROGRAMS AND BLOCKS

1 SYNTAX 1
2 SEMANTICS 1

2 ALGOL DECLARATIONS

1 SYNTAX 3
2 RESTRICTIONS 4
3 EXAMPLES 5
4 SEMANTICS 5
5 SEPARATELY COMPILED PROCEDURES 10

3 ALGOL STATEMENTS

1 SYNTAX 13
2 SEMANTICS 14

4 ALGOL EXPRESSIONS

1 SYNTAX 20
2 TYPE CONVERSION 21
3 SEMANTICS 22

5 ASSEMBLY LANGUAGE STATEMENTS

1 SYNTAX 26
2 SEMANTICS 26

6 BACKTRACKING

1 INTRODUCTION 29
2 SYNTAX 29
3 SEMANTICS 29

7 INPUT/OUTPUT ROUTINES

1 EXECUTION TIME ROUTINES IN GENERAL 31
2 I/O CHANNELS AND FILES 31
3 BREAK CHARACTERS 33
4 I/O ROUTINES 35
5 TELETYPE AND PSEUDO-TELETYPE ROUTINES 38

8 EXECUTION TIME ROUTINES

1 TYPE CONVERSION ROUTINES 40
2 STRING MANIPULATION ROUTINES 41
3 LIBERATION-FROM-SAIL ROUTINES 41
4 BYTE MANIPULATION ROUTINES 42
5 OTHER USEFUL ROUTINES 43

9 MACROS AND CONDITIONAL COMPILATION

1 SYNTAX 45
2 DELIMITERS 46
3 MACROS 46
4 MACROS WITH PARAMETERS 48
5 CONDITIONAL COMPILATION 49
6 TYPE DETERMINATION AT COMPILE TIME 49
7 MISCELANEOUS FEATURES 50

I. 0 LEAP DATA TYPES

1 INTRODUCTION 51
2 SYNTAX 51
3 SEMANTICS 52

11 LEAP STATEMENTS

1 SYNTAX 55
2 RESTRICTIONS 56
3 SEMANTICS 56
4 SEARCHING THE ASSOCIATIVE STORE 57

12 LEAP EXPRESSIONS

1 SYNTAX 63
2 SEMANTICS 64

13 PROCESSES

1 INTRODUCTION 67
2 SYNTAX 67
3 SEMANTICS 67

14 EVENTS

1 SYNTAX 72
2 INTRODUCTION 72
3 SAIL DEFINED CAUSE AND INTERROGATE 72
4 USER DEFINED CAUSE AND INTERROGATE 73

15 PROCEDURE VARIABLES

1 SYNTAX 76
2 SEMANTICS 76

16 INTERRUPTS

1 INTRODUCTION 78
2 IMMEDIATE INTERRUPTS 78
3 DEFERRED INTERRUPTS 80
4 MORE COMPLICATED DEFERRED INTERRUPTS 80

17 LEAP AND PROCESS RUNTIMES

1 TYPES AND TYPE CONVERSION 83
2 MAKE AND ERASE BREAKPOINTS 83
3 PNAME RUNTIMES 84
4 OTHER USEFUL RUNTIMES 84
5 GENERAL PROCESS RUNTIMES 85
6 RUNTIMES FOR USER CAUSE AND INTERROGATE PROCEDURES 86

18 BASIC CONSTRUCTS

1 SYNTAX 88
2 SEMANTICS 88

19 USING SAIL

1 FOR BEGINNERS 90
2 THE COMPLETE USE OF SAIL 90
3 COMPILING SAIL PROGRAMS 90
4 LOADING SAIL PROGRAMS 93
5 STARTING SAIL PROGRAMS 93
6 STORAGE REALLOCATION WITH THE REENTER COMMAND 94

20 DEBUGGING SAIL PROGRAMS

1 ERROR MESSAGES 95
2 DEBUGGING 96

A APPENDICES

1 TYPE CONVERSION 98
2 SAIL RESERVED WORDS 99
3 SAIL PRE-DECLARED IDENTIFIERS 99
4 CHARACTER-IDENTIFIER EQUIVALENCES 99
5 PARAMETERS TO THE OPEN FUNCTION 99
6 BREAKSET MODES 100
7 MTAPE COMMANDS 100
8 COMPILE SWITCHES 100
9 VALID RESPONSES TO ERROR MESSAGES 100
10 ERROR CODES 1Ø1
11 INDICES FOR INTERRUPTS 102
12 BIT NAMES FOR PROCESS CONSTRUCTS 103
13 STATEMENT COUNTER SYSTEM 104
14 ARRAY IMPLEMENTATION 106
15 STRING IMPLEMENTATION 107
16 PROCEDURE IMPLEMENTATION 107

R REFERENCES 110

I INDEX 110

SECTION 1

PROGRAMS AND BLOCKS

1.1 - SYNTAX

```
<program>
    ::= <block>

<block>
    ::= <block-head> ; <compound-tail>

<block-head>
    ::= BEGIN <declaration>
    ::= BEGIN <block-name> <declaration>
    ::= <block-head> ; <declaration>

<compound-tail>
    ::= <statement> END
    ::= <statement> END <block-name>
    ::= <statement> ; <compound-tail>

<compound-statement>
    ::= BEGIN <compound-tail>
    ::= BEGIN <block-name> <compound-tail>

<statement>
    ::= <block>
    ::= <compound-statement>
    ::= <require_specification>
    ::= <assignment>
    ::= <swap-statement>
    ::= <conditional_statement>
    ::= <If-statement>
    ::= <go-to-statement>
    ::= <for-statement>
    ::= <while-statement>
    ::= <do-statement>
    ::= <case-statement>
    ::= <return-statement>
    ::= <done-statement>
    ::= <next-statement>
    ::= <continue_statement>
    ::= <procedure-statement>
    ::= <safety-statement>
    ::= <backtracking-statement>
    ::= <code-block>
    ::= <leap-statement>
    ::= <process-statement>
    ::= <event-statement>
    ::= <string-constant> <statement>
    ::= <label-identifier> : <statement>
    ::= <empty>
```

1.2 - SEMANTICS

DECLARATIONS

SAIL programs are organized in the traditional block structure of ALGOL-60.

Declarations serve to define the data types and dimensions of simple and subscripted (array) variables (arithmetic variables, strings, sets, and items). They are also used to describe procedures (subroutines) and name program labels.

Any identifier referred to in a program must be described in some declaration. An identifier may only be referenced by statements within the scope (see page 5) of its declaration.

STATEMENTS

As in ALGOL, the statement is the fundamental unit of operation in the SAIL language. Since a statement within a block or compound statement may itself be a block or compound statement, the concept of statement must be understood recursively.

The block representing the program is known as the "outer block". All blocks internal to this one will be referred to as "inner blocks".

BLOCK NAMES

The block name construct is used to describe the block structure of a SAIL program to a symbolic debugging routine (see page 96). The name of the outer block becomes the title of the binary output file (not necesarily the file name). In addition, if a block name is used following an END, the compiler compares it with the block name which followed the corresponding BEGIN. A mismatch is reported to the user as evidence of a missing (extra) BEGIN or END somewhere.

The <string-constant> <statement> construct is equivalent in action to the <statement> alone; that is, the string constant serves only as a comment.

EXAMPLES

Given:
S is a statement,
Sc is a Compound Statement,
D is a Declaration,
B is a Block.

Then:

```
(Sc)   BEGIN S; S; S; . . . ;S END

(Sc)   BEGIN "SORT" S; S; . . . ;S END "SORT"

(B)    BEGIN D; D; D; . . . ; S; S; S; . . . ;S END

(B)    BEGIN "ENTER NEW INFO" D; D; . . . ;
                                S; . . . ;S END
```

are syntactically valid SAIL constructs.

SECTION 2

ALGOL DECLARATIONS

2.1 - SYNTAX

```
<id_list>
     ::= <ident if ier>
     ::= <identifier> , <id-list>

<declaration>
     ::= <type-declaration>
     ::= <array-declaration>
     ::= <preload_specification>
     ::= <label-declaration>
     ::= <procedure-declaration>
     ::= <synonym-declaration>
     ::= <require-specification>
     ::= <context-declaration>
     ::= <leap-declaration>
     ::= <protect-acs declaration>
     ::= <cleanup-declaration>

<simple-type>
     ::= REAL
     ::= INTEGER
     ::= BOOLEAN
     ::= STRING

<type-qualifier>
     ::= EXTERNAL
     ::= INTERNAL
     ::= SAFE
     ::= FORWARD
     ::= RECURSIVE
     ::= FORTRAN
     ::= SIMPLE
     ::= OWN
     ::= SHORT

<type-declaration>
     ::= <simple-type> <id_list>
     ::= <type-qualifier> <type-declaration>

<array-declaration>
     ::= <simple-type> ARRAY <array-list>
     ::= <type-qualifier> <array declaration>

<array-list>
     ::= <array-segment>
     ::= <array-list> , <array-segment>

<array-segment>
     ::= <id_list> [ <bound_pair_list> 1

<bound-pair-list>
     ::= <bound-pair>
     ::= <bound-pair-list> , <bound_pair>

<bound-pair>
     ::= <lower-bound>   <upper-bound>

<lower-bound>
     ::= <algebraic-expression>

<upper-bound>
     ::= <algebraic-expression>

<preload_specification>
     ::= PRELOAD_WITH <preload_list>

<preload_list>
     ::= <preload_element>
     ::= <preload_list> , <preload element>

<preload_element>
     ::= <expression>
     ::= [expression] <expression>

<label-declaration>
     ::= LABEL <id_list>
```

<procedure_declaration>
::= PROCEDURE <identifier> <procedure-head>
<procedure-body>
::= <simple_type> PROCEDURE <identifier>
<procedure_head> <procedure-body>
::= <type_qualifier> <procedure-declaration>

<procedure-head>
::= <empty>
::= (<formal_param_decl>)

<procedure-body>
::= <empty>
::= ; <statement>

<formal_param_decl>
::= <formal_parameter_list>
::= <formal-parameter-list> ;
<formal_param_decl>

<formal_parameter_list>
::= <formal-type> <id_list>

<formal-type>
::= <simple_formal_type>
::= REFERENCE <simple-formal-type>
::= VALUE <simple_formal_type>

<simple-formal-type>
::= <simple_type>
::= <simple_type> ARRAY
::= <simple_type> PROCEDURE

<synonym-declaration>
::= LET <synonym-list>

<synonym_list>
::= <synonym>
::= <synonym-list> , <synonym>

<synonym>
::= <identifier> = <reserved-word>

<cleanup-declaration>
::= CLEANUP <procedure_identifier_list>

<require_specification>
::= REQUIRE <require-list>

<require-list>
::= <require-element>
::= <require-list> , <require-element>

<require-element>
::= <constant> <require_spec>
::= <procedure-name> INITIALIZATION

<require_spec>
::= STRING-SPACE
::= SYSTEM-PDL
::= STRING-PDL
::= ARRAY-PDL
::= NEW-ITEMS
::= PNAMES
::= LOAD-MODULE
::= LIBRARY
::= SOURCE-FILE
::= SEGMENT-FILE
::= SEGMENT-NAME
::= POLLING-POINTS
::= VERSION
::= ERROR-MODES
::= DELIMITERS
::= BUCKETS
::= MESSAGE

2.2 - RESTRICTIONS

For simplicity, the type-qualifiers are listed in only one syntactic class. Although their uses are always valid when placed according to the above syntax, most of them only have meaning when applied to particular subsets of these productions:

SAFE is only meaningful in array declarations.

INTERNAL/EXTERNAL have no meaning in formal parameter declarations.

SIMPLE, FORWARD, RECURSIVE, and FORTRAN have meaning only in procedure type specifications.

SHORT has meaning only when applied to INTEGER or REAL entities.

For array declarations in the outer block substitute

<constant-expression> for <algebraic-expression> in the productions for <lower-bound> and <upper-bound>.

A label must be declared in the Innermost block in which the statement being labeled appears (more information, page 15). The syntax for procedure declarations requires semantic embellishment (see page 7) in order to make total sense. In particular, a procedure body may be empty only in a restricted class of declarations.

2.3 - EXAMPLES

Let I,J,K,L,X,Y, and P be identifiers, S a statement:

```
(<type_declaration>)
        INTEGER I,J,K
        EXTERNAL REAL X,Y
        INTERNAL STRING K

(<array_declaration>)
        INTEGER ARRAY X[Ø:1Ø,Ø:1Ø]
        REAL ARRAY Y[X:P(L)]; Comment illegal
          in outer block
        STRING ARRAY I[Ø:IF BIG THEN 30 ELSE 3]

(<label-declaration>)
        LABEL L,X,Y

(<procedure declaration>)
        PROCEDURE P; S
        PROCEDURE P(INTEGER I,J;
          REFERENCE REAL X; REAL Y); S
        INTEGER PROCEDURE P (REAL PROCEDURE L;
          STRING I,J; INTEGER ARRAY K); S
        EXTERNAL PROCEDURE P(REAL X)
        FORWARD INTEGER PROCEDURE X(INTEGER I)
        FORTRAN REAL PROCEDURE SIN
```

Note that these sample declarations are all given without the semicolons which would normally separate them from the surrounding declarations and statements. Here is a sample block to bring it all together (again, let S be any statement, D any declaration, and other identifiers as above):

```
BEGIN "SAMPLE BLOCK"
 INTEGER I,J,K;
 REAL X,Y;
 STRING A;
 INTEGER PROCEDURE P(REFERENCE REAL X);
  BEGIN "P"
   D; D; D; . . . ;S; . . . ; S
  END "P";

 REAL ARRAY DIPHTHONGS[Ø:1Ø,1:1ØØ];

 S; S; S; S
END "SAMPLE BLOCK"
```

2.4 - SEMANTICS

SCOPE OF DECLARATIONS

Every block automatically introduces a new level of nomenclature. Any identifier declared in a block's head is said to be LOCAL to that block. This means that:

a. The entity represented by this identifier inside the block has no existence outside the block.

b. Any entity represented by the same identifier outside the block is completely inaccessible (unless it has been passed as a parameter) inside the block.

An identifier occurring within an inner block and not declared within that block will be nonlocal (global) to it; that is, the identifier will represent the same entity inside the block and in the block or blocks within which it is nested, up to and including the level in which the identifier is declared.

The Scope of an entity is the set of blocks in which the entity is represented, using the above rules, by its identifier. An entity may not be referenced by any statement outside its scope.

TYPE QUALIFIERS

An array, variable, or procedure declared OWN will behave as if it were declared globally to the current procedure: the OWN type qualifier on a variable, etc. declared in a block not nested inside a procedure declaration will have no effect. This means that in a second call of a procedure with OWN locals (or a recursive call) the OWN variables will not be reinitialized they will have the values that they had when the first call of the procedure finished. Furthermore, OWN arrays, etc. will not be deallocated upon exiting the procedure they are declared in.

INTERNAL and EXTERNAL procedures, variables, etc. let one link programs that are loaded together, but were compiled separately. See page 1Ø for more information.

RECURSIVE, SHORT, FORTRAN, FORWARD, SIMPLE, and SAFE will be explained when the data types they modify are discussed.

NUMERIC DECLARATIONS

Identifiers which appear in type declarations with types REAL or INTEGER can subsequently be used to refer to numeric variables. An Integer variable may take on values from -2↑35 to 2↑35-1 (-2↑26 to 2↑26-1 for SHORT INTEGERS). A Real variable may take on positive and negative values from about 1Ø↑-38 to 1Ø↑38 with a precision of 27 bits (same range for SHORT REALs as for SHORT INTEGERs. REAL and

INTEGER variables (and constants) may be used in the same arithmetic expressrons. type conversions are carried out automatically (see page 21 below) when necessary.

The advantage of SHORT reals and Integers is that the conversion from integer to real is sped by a factor of 8 if either the integer or the real is SHORT. See page 2 1 for more information.

The BOOLEAN type is identical to INTEGER. As you will see, BOOLEAN and algebraic expressions are really equivalent syntactically. The syntactic context in which they appear determines their meaning. Non-zero integers correspond to TRUE and 0 corresponds to FALSE The declarator BOOLEAN is included for progratn clarity.

STRING DECLARATIONS

A variable defined in a String declaration is a two-word descriptor containing the information necessary to represent a SAIL character string.

A String may be thought of as a variable-length, one-dimensional array of 7-bit ASCII characters. Its descriptor contains a character count and a byte pointer to the first character (see page 1Ø7). Strings originate as constants at compile time (page 89), as the result of a String INPUT operation from some device (see page 35), or from the concatenation or decomposition of already existing strings (see page 24 and page 24)

When strings appear in arithmetic operations or vice-versa, a somewhat arbitrary conversion is performed to obtain the proper type (by arbitrary we do not mean to imply random -- see page 21). For this reason arithmetic and String variables are referred to as "algebraic variables" and their corresponding expressions are called "algebraic expressions" (to differentiate them them from the variables and expressions of LEAP -- see page 51)

ARRAY DECLARATIONS

in general, any data type which is applicable to a simple variable may be applied in an Array declaration to an array of variables. The entity represented by the name of an Array, qualified with subscript expressions to locate a particular eiement (e.g. A[I,J]) behaves in every way like a simple variable. Therefore, in the future we shall refer to both simple variables and single elements of Arrays (subscripted variables) as "variables" The formal syntax for <variable> can be found on page 88.

For an Array which is not qualified by the SAFE attribute, nor had a NOW-SAFE statement done on it (Now_Safe - see page 19), each subscript will be checked to ensure that it falls within the lower and upper bounds given for the dimension it specifies. Subscripts outside the bounds trigger an error message and job abortion. The SAFE declaration inhibits this checking, resulting in faster, smaller, and bolder code.

There is no limit to the number of dimensions allowed for an Array. However, the efficiency of Array references tends to decrease for large dimensions Avoid large dimensionality if it is not necessary.

OWN Arrays are available in part. They must be declared with constant bounds, since fixed storage is allocated for these Arrays. They are NOT initialized when the program is started or restarted (except in preloaded Arrays, see page 6). A certain degree of extra efficiency is possible in accessing these Arrays, since they may be assigned absolute core locations by the compiler, eliminating some of the address arithmetic. Constant bounds always add a little efficiency, even in inner blocks. Arrays declared in the outer block must have constant bounds, since no variable may yet have been assigned a value. They are thus automatically made OWN. For more details concerning the internal structure of Arrays see page 96 and page 106.

PRELOAD SPECIFICATIONS

Any OWN arithmetic or String Array may be "pre-loaded" at compile time with constant information by preceding its declaration with a <preload_specification>. This specification gives the values which are to be placed in consecutive core locations of the Arrays declared immediately following the <preload_specification>. "Immediately", in this case, means all identifiers up to and including one which is followed by bound-pair-list brackets (e.g. in REAL ARRAY X,Y,Z[Ø:1Ø],W[1:5]; -- preloads X,Y, and Z, not W). It is the user's responsibility to guarantee that the proper values will be obtained under the subscript mapping, namely: arrays are stored by rows; if A[I,J] is stored in location 10000, then A[I,J+1] is stored in location 10001.

The current values of pre-loaded Arrays will not be lost by restarting the program: they will not be re-initialized or re-preloaded. For preloaded String Arrays, this means you may have invalid string descriptors after a restart: the contents of the array will not change over the restart, but string space will change, leaving the elements of the array pointing off into the boondocks

Algebraic type conversions will be performed at compile-time to provide values of the proper types to pre-loaded Arrays All expressions in these specifications must be constant expressions -- that is, they must contain only constants and algebraic operators. The compiler will not allow you to fill an Array beyond Its capacity. You may, however, provide a number of elements less than the total size of the

Array: remaining elements will be set to zero or to the null string.

Example,

```
PRELOAD_WITH[5] 0, 3, 4, [4] 6, 2;
INTEGER ARRAY TABL[1:4,1:3];
```

The first five elements of TABL will be initialized to 0 (bracketed number is used as a repeat argument). The next two elements will be 3 and 4, followed by four 6's and a 2. The array will look like this.

```
   1 2 3
   - - - - - -
1 | 0 0 0
2 | 0 0 3
3 | 4 6 6
4 | 6 6 2
```

PROCEDURE DECLARATIONS

If a Procedure is typed, it may return a value (see page 17) of the specified type. If formal parameters are specified, they -must be supplied with actual parameters in a one to one correspondence when they are called (see page 24 and page 18).

FORMAL PARAMETERS

Formal parameters, when specified, provide information to the body (executable portion) of the Procedure about the kinds of values which will be provided as actual parameters in the call. The type and complexity (simple or Array) are specified here. In addition, the formal parameter indicates whether the value (VALUE) or address (REFERENCE) of the actual parameter will be supplied. If the address is supplied, the variable whose identifer is given as an actual parameter may be changed by the Procedure. This is not the case if the value is given.

To pass a PROCEDURE by value has no readily determined meaning. ARRAYs passed by value (requiring a complete copy operation) are not implemented Therefore these cases are noted as errors by the compiler,

The proper use of actual parameters is further discussed on page 18 and page 24.

FORWARD PROCEDURE DECLARATIONS

A Procedure's type and parameters must be described before the Procedure may be called. Normally this is accomplished by specifying the procedure declaration in the head of some block containing the call. If, however, it is necessary to have two Procedures, declared in some block head, which are both accessible to statements in the compound tail of that block and to each other, the FORWARD construct permits the definition of the parameter information for one of these Procedures in advance of its declaration. The Procedure body must be empty in a forward procedure declaration When the body of the Procedure described in the forward declaration is actually declared, the types of the Procedure and of its parameters must be identical in both declarations. The declarations must appear at the same level (within the same block head).

Example:

```
BEGIN "NEED FORWARD"
   FORWARD INTEGER PROCEDURE T1(INTEGER I);
          COMMENT PARAMS DESCRIBED;
   INTEGER PROCEDURE T2(INTEGER J);
          RETURN (T1(J)+3);
          COMMENT CALL T1;
   INTEGER PROCEDURE TI (INTEGER I);
          COMMENT ACTUALLY DEFINE T1;
          RETURN (IF I=15 THEN I
                 ELSE T2(I-1));
          COMMENT CALLS T2;

   ...

   K←T1(L); . . . ; L←T2(K); . . .

END "NEED FORWARD";
```

Notice that the forward declaration is required only because BOTH Procedures are called in the body of the block. These procedures should also be declared RECURSIVE if recursive entrance is likely. If only T1 were called from statements within the block, this example could be implemented as:

```
BEGIN "NO FORWARD"
   RECURSIVE INTEGER PROCEDURE T1 (INTEGER I);
   BEGIN
          INTEGER PROCEDURE T2(J);
                 RETURN (T1(J)+3);
          RETURN( IF I=15 THEN I
                        ELSE T2(I-1));
   END "T1";
   ...
   K←T1(L);

...

END "NO FORWARD";
```

RECURSIVE PROCEDURES

If a Procedure is to be entered recursively, the compiler must be instructed to provide code for allocating new local variables when the Procedure is called and deallocating them when it returns. Use the type-qualifier RECURSIVE in the declaration of any recursive Procedure.

The cotnpiler can produce much more efficient code for non-recursive Procedures than for recursive ones. We feel that this gain in efficiency merits the necessity for declaring Procedures to be recursive.

If a Procedure which has not been declared recursive is called recursively, all its local variables (and

temporary storage locations assrgned by the compiler) will behave as if they were global to the Procedure -- they will not be reinitialized, and when the recursive call is complete, the locals of the calling procedure will reflect the changes made to them during the recursive call. Otherwise, no ill effects should be observed.

SIMPLE PROCEDURES
Standard procedures contain a short prologue that sets up some links on the stack and a descriptor that is used by the storage allocation system, the go to solver, and some other routines. For most procedures, this overhead is insignificant. However, for small procedures that just do a few simple statements and exit, this overhead is excessive and unneeded. To skip the prologue, just include SIMPLE in the attribute list for the procedure. RESTRICTIONS,

1. Simple procedures may not be Recursive.
2. ARRAY locals must be OWN
3. Set and List locals must be OWN (Sets and list are part of Leap, page 5 1).
4. Procedures declared local to a simpie procedure must also be of of type SIMPLE, and may not reference any of the parameters of the outer simple procedure.
5. One may not GO TO a statement outside the body of the simple procedure.

EXTERNAL PROCEDURES
A file compiled by SAIL represents either a "main" program or a collection of independent procedures to be called by the main program. The method for preparing such a collection of Procedures is described in page 10 The EXTERNAL and FORTRAN type-qualifiers allow description of the types of these Procedures and their parameters. An EXTERNAL or FORTRAN procedure declaration, like the FORWARD declaration, does not include a procedure body. Both declarations Instead result in requests to the loader to provide the addresses of these Procedures to ail statements which call them. This means that an EXTERNAL Procedure declaration (or the declaration of any External Identifier) may be placed within any block head, thereby controlling the scope of this External identifier within this program.

Any SAIL Procedure which is referenced via these external declarations must be an INTERNAL Procedure. That is, the type-qualifier INTERNAL must appear in the actual declaration of the Procedure. Again, see page 10.

The type-qualifier FORTRAN is used to describe the type and name of an external Procedure which is to be called using a DEC Fortran calling sequence. All parameters to Fortran Procedures are by reference In fact, the procedure head part of the declaration need not be Included unless the types expected by the Procedure differ from those provided by the actual parameters--the number of parameters supplied, and their types, are presumed correct. Fortran Procedures are automatically External Procedures. See page 9, page 18, page 24 for more information about Fortran Procedures.

Example:

```
FORTRAN PROCEDURE MAX;
Y←MAX(X,Z);
```

PARAMETRIC PROCEDURES
The calling conventions for Procedures with Procedures as arguments, and for the execution of these parametric Procedures, are described on page 18 and page 24. Any Procedure PP which is to be used as a parameter to another Procedure CP must not have any Procedure or array parameters, or any parameters called by value. In other words, PP may only have simple reference parameters. The number of parameters supplied in a call on PP within CP, and their types, will be presumed correct, and should not be specified in the procedure head.

Example:

```
PROCEDURE CP (INTEGER PROCEDURE FP);
   BEGIN INTEGER A,I; REAL X;
   ...
   A←FP(I,X); COMMENT I AND X PASSED BY
         REFERENCE, NO TYPE CONVERSION;
   END "CP";

INTEGER PROCEDURE FP (REFERENCE INTEGER J;
         REFERENCE REAL Y);
   BEGIN
   ...
   END "PP"

 ...

CP(PP);
```

DEFAULTS IN PROCEDURE DECLARATIONS
If no VALUE or REFERENCE qualification appears in the description, the following qualifications are assumed,

VALUE	Simple Integer, String, or Real Variables.
REFERENCE	Arrays, Contexts and Procedures.

RESTRICTIONS ON PROCEDURE DECLARATIONS

Fortran Procedures can not handle String parameters. Nor can a Fortran Procedure return a string as a result.

2) Labels may never be passed as arguments to Procedures.

3) Procedures may not have the type "CONTEXT".

4) Context parameters must always be passed by reference.

ALLOCATION AND DEALLOCATION

All simple variables (integer, real, string, boolean) are allocated at compile time. Non-own simple variables that are local to a recursive procedure are an exception to this and are allocated (on the stack) upon instantiation of the procedure; they are deallocated when the instantiation is terminated.

All outer block arrays are allocated at compile time. All Own arrays are allocated at compile time. All other arrays are allocated when the block of their definition is entered, and deallocated when it is exited.

INITIALIZATION AND REINITIALIZATION

Upon allocation, everything is initialized to 0 or the NULL string (except preloaded arrays, which are initialized to their the values of their PRELOAD). Nothing is reinitialized unless the program is restarted by typing ↑C and REEnter. This lack of reinitiaiization is notrceable when one enters a block for the second time, and that block is not the body of a recursive procedure. For example,

```
STRING PROCEDURE READIN;
BEGIN
    INTEGER CHANNEL, BRTAB;
    IF BRTAB=0 THEN BRTAB ←INIT(CHANNEL);
    RETURN(INPUT(CHANNEL, BRTAB ));
END;
```

will return a string from an input operation with every call. However, on the first call, it will do some initialization of the I/O channel because BRTAB is 0 then, whereas it is not for any of the other calls. If READIN were a recursive procedure, CHANNEL and BRTAB would be allocated and hence initialized with every call.

When one REEenters a program, some things are reinitilized and some are not. Namely, strings and non-preloaded arrays will be reinitialized, but simple vat-tables will not. Preloaded arrays will not be re-preloaded.

SYNONYMS

The Sail Synonym permits one to declare any identifier to act as a reserved word. The effect of the reserved word is not changed: it may be used as well as the new identifier. Synonyms follow the same scope rules that identifiers used for variables, arrays, etc. do.

Since Sail permits one to declare almost any reserved word to be an identifier for variables, procedures, etc. (see about restrictions on identifiers, page 89), synonyms are used to keep the effect of the reserved word available. For example,

```
LET BEG = BEGIN;
PROCEDURE BEGIN;
  BEG
    ......

  END;

...
IF OK THEN BEGIN;
...
```

CLEANUP DECLARATIONS

The CLEANUP declaration requires a list of procedure names following the "CLEANUP" token. Each procedure specified must be SIMPLE and have no formal parameters. The specified procedures will be called at the exit of the block that the CLEANUP declaration occurs in. They will be called in the order of their appearance on the list, and before any of the variables of the block are deallocated. NOTE: If the block is part of a process (see about processes, page 67) that is being terminated, the cleanup procedures will be called before the terminate is completed.

Cleanup procedures are normally used in connection with processes to "cleanup" a block by terminating the processes dependent on that block (it is an error to leave processes active that depended on an exited block).

REQUIREMENTS

The user may, using the REQUIRE construct, specify to the compiler conditions which are required to be true of the execution-time environment of his programs. All requirements are legal at either declaration or statement level. The requirements fall into three classifications, described as follows:

Group 1 -- Space requirements -- STRING-SPACE, SYSTEM-PDL, etc.

The inclusion of the specification "REQUIRE 1000 STRING-SPACE" will ensure that at least 1000 words of storage will be available for storing Strings when the program is run. Similar provisions are made for various push-down stacks used by the execution-time routines and the compiled code. If a parameter is specified twice, or if separately compiled procedures are loaded (see page 10), the sum of all such

specifications will be used. These parameters could also b e typed to the loaded program just before execution (see page 94), but it is often more convenient to specify differences from the standard sizes in the source program. Use these specifications only if messages from the running program indicate that the standard allocations are not sufficient.

Group 2 -- Other files -- LOAD-MODULE, LIBRARY, SOURCE-FILE, etc.

The inclusion of the specification REQUIRE "PROCS1" LOAD-MODULE, "HELIB[1,3]" LIBRARY; would inform the Loader that the file PROCS1.REL must be loaded and the library HELIB.REL[1,3] searched whenever the program containing the specification is loaded. The parameter for both features should be a string constant of one of the above forms. The device DSK, and file extension .REL are the only values permitted for these entries, and are therefore assumed.

LOAD-MODULES (.REL files to be loaded) may themselves contain requests for other LOAD-MODULES and LIBRARYs. LIBRARYs may only contain requests for other LIBRARYs. Duplicate specifications are in general merged into single requests (if a file is requested twice, it will be loaded only once).

SAIL automatically places a request for the library "SYS:LIBSAn" in each main program, where n is the version number of the current Sail library of runtime routines.

The inclusion of REQUIRE "SYS:PREAMB.SAI" SOURCE-FILE will cause the compiler to save the state of the current input file, then begin scanning from PREAMB. When PREAMB is exhausted, SAIL will resume scanning the original file on the line directly following the REQUIRE. SOURCE-FILEs may be nested to a depth of about 10 levels.

Restrictions: A SOURCE-FILE request must be followed by a semicolon (only one per REQUIREment), and must be the last text on the line in which it appears SOURCE-FILE switching must not be specified from within a DEFINE body (see page 46).

The SEGMENT-NAME, SEGMENT-FILE specifications are currently applicable only to the Stanford "global model" users of SAIL. They allow specification of the name of a special non-sharable "HISEG", and the name of the file used to create this HISEG. These specifications may, like the space REQUIREments, be overridden by using the system REENTER command (see page 94).

Group 3 -- other - INITIALIZATION, VERSION

Before the execution of a program, Sail runs through an initialization routine. The user can specify things that he wants done at initialization time by declaring a Procedure without arguments, then saying

REQUIRE procedure-name INITIALIZATION.

The narmcd procedure will be run called as the first executable statement in the outer block of the program (even if the REQUIRE appeared in a Source or REL file). Require-initialization procedures will be run in the order in which they were Required. WARNING: you should not Require initialization of a procedure which is declared inside another procedure.

REQUIRE n VERSION in a non-zero integer) will flag the resultant RELfile as version n. When a program loaded from several such RELfiles is started, the Sail allocatin code will verify that all specified versions are equal. A non-fatal error message is generated if any disagree. As much as will fit of the version number is also stored in lh(JOBVER), where JOBVER is location 137.

Other requirements: PNAMES - see page 84; POLLING POINTS - see page 70; DELIMITERS - see page 46; BUCKETS - see page 58; NEW-ITEMS - see page 64; MESSAGE - see page 50; ERROR-MODE - see page 95.

COMMENT: You have probably noticed that a great deal of prior knowledge is required for proper understanding of this section. For more information about storage allocation, see page 94 below. The form and use of .REL files and libraries are described in "The Stanford A-I Project Monitor Manual" [Moorer] and [Weiher].

2.5 - SEPARATELY COMPILED PROCEDURES

When a program becomes extremely large it becomes useful to break the program up into several files which can be compiled separately. This can be done in SAIL by preparing one file as a main program, and one or more other files as programs each of which contains one or more procedures to be called by the main program The main program must contain EXTERNAL declarations for each of the procedures declared in the other files (EXTERNAL declarations have no procedure body). The non-main program files must have the following characteristics:

1) All procedures to be called from the main program (or procedures in other files) must be qualified with the INTERNAL attribute when they are declared. External procedure declarations with headings identical to those of the actual declarations must appear in all those programs which call these procedures.

2) These internal procedures must be uniquely identifiable by the first six characters of their Identifiers. In general, any two internal procedure names (or any other Internal variables in the same core image) with the same first six characters will cause incorrect linkages when the programs are loaded.

3) The reserved word ENTRY, followed by a semi-colon must be the first item in the program (preceding even the BEGIN for its outer block). No starting address will be issued for a program containing an Entry Specification. Since no starting address is present for this file, entry to code within it may only be to the procedures it contains. The statements in the outer block, if any, can never be executed.

4) Should you desire your separatedly compiled procedures to be collected into a user library, include a list of their identifiers between the ENTRY and the semi-colon of the Entry Specification of the program containing those procedure declarations. The format of libraries is described in [Weiher]. The identifier(s) appearing in the entry list may be any valid identifiers, but usually they will be the names of the procedures contained in the file. No checking is done to see if entry identifiers are ever really declared in the body of the program.

5) Any variables (simple or array) which appear in the outer block of a Separately Compiled Procedure program will be global to the procedures in this program, but not available to the main program (unless they are themselves connected connected to the main program by Internal/External declarations -- see below). Arithmetic arrays in these outer blocks will always be zero when the program is first loaded, but will never be cleared as others are by restarting your program (see reinitialization, page 9).

Any variable, procedure or label may contain the attribute INTERNAL or EXTERNAL in its declaration (ITEMS may not -- items are part of leap, page 5 1). The INTERNAL attribute does not affect the storage assignment of the entity it represents, nor does it have any effect on the behavior of the entity (or the scope of its Identifier) in the file wherein it appears. However, its address and (the first six characters of) its name are made available to the loader for satisfying External requests.

No space is ever allocated for an External declaration. Instead, a list of references to each External identifier is made by the compiler. This list is passed to the loader along with the first six characters of the identifier name. When an Internal name matching it is found during loading, its associated address is placed in each of the instructions mentioned on the list. No program inefficiency at all results from External/Internal linkages (belay that -- references to External arrays are sometimes more inefficient).

The entity finally represented by an External identifier is only accessible within the scope of the External declaration.

FORTRAN PROCEDURES

For a program written in DEC FORTRAN IV to run in the SAIL environment, the following restrictions must be observed:

1) It must be a SUBROUTINE or FUNCTION, not a main program.

2) It must not execute any FORTRAN I/O calls. The UUO structures of the two languages are not compatable.

3) It must be declared as a Fortran Procedure (see page 19) in the SAIL program which calls it.

The type bits required in the argument addresses for Fortran arguments are passed correctly to these routines.

The SAIL compiler will not produce a procedure to be called from FORTRAN.

ASSEMBLY LANGUAGE PROCEDURES

The following rules should be observed:

1) The ENTRY, INTERNAL, and EXTERNAL pseudo-ops should be used to obtain linkages for procedure names and "global" identifiers (remember that only six characters are used for these linkage names.

2) Accumulators F (currently '12), P (currently '17) and SP ('16) should be preserved over function calls. P may be used as a push-down pointer for arithmetic values and return addresses. SP is the string stack pointer. String results are returned on this stack. Arithmetic results are returned in AC 1.

3) Those who wish to provide their own UUO handlers or to increase their core size should read the relevant sections of the Implementation manual.

There are no other known processors which will produce SAIL-compatible programs. In particular, the LISP 1 6 system, by its very nature, contains storage allocation conflicts which are difficult to resolve. If a great need for thrs kind of compatibility develops it can be provided.

SECTION 3

ALGOL STATEMENTS

3.1 - SYNTAX

<assignment-statement>
 ::= <algebraic-variable> ←
 <algebraic_expression>

<swap-statement>
 ::= <variable> ↔ <variable>

<conditional-statement>
 ::= <If-statement>
 ::= <if_statement> ELSE <statement>

<If-statement>
 ::= IF <boolean-expression> THEN <statement>

<go-to-statement>
 ::= GO TO <label-identifier>
 ::= GOTO <label-identifier>
 ::= GO <label-identifier>

<label-identifier>
 ::= <identifier>

<for-statement>
 ::= FOR <algebraic-variable> ← <for-list> DO
 <statement>
 ::= NEEDNEXT <for-statement>

<for-list>
 ::= <for-list-element>
 ::= <for-list> , <for-list-element>

<for-list-element>
 ::= <algebraic-expression>
 ::= <algebraic-expression> STEP
 <algebraic-expression> UNTIL
 <algebraic-expression>
 ::= <algebraic-expression> STEP
 <algebraic-expression> WHILE
 <boolean-expression>

<while-statement>
 ::= WHILE <boolean-expression> DO
 <statement>
 ::= NEEDNEXT <while-statement>

<do-statement>
 ::= DO <statement> UNTIL
 <boolean-expression>
 ::= NEEDNEXT <do-statement>

<case-statement>
 ::= <case-statement-head> <statement-list>
 <case-statement-tail>
 ::= <case-statement-head>
 <numbered-state-list>
 <case-statement-tail>

<case-statement-head>
 ::= CASE <algebraic-expression> OF BEGIN
 ::= CASE <algebraic-expression> OF BEGIN
 <block-name>

<case-statement-tail>
 ::= END
 ::= END <block-name>

<statement-list>
 ::= <statement>
 ::= <statement-list> ; <statement>

<numbered-state-list>
 ::= [<integer_constant>] <statement>
 ::= <numbered-state-list> ;
 [<integer-constant>] <statement>

<return-statement>
 ::= RETURN
 ::= RETURN (<expression>)

<done-statement>
 ::= DONE

```
     ::= DONE <block-name>

<next_statement>
     ::= NEXT
     ::= NEXT <block-name>

<continue-statement>
     ::= CONTINUE
     ::= CONTINUE <block-name>

<procedure-statement>
     ::= <procedure-call>

<procedure-call>
     ::= <procedure-ident if ier>
     ::= <procedure_ident if ier> (
          <actual-parameter-list> )

<actual-parameter-list>
     ::= <actual-parameter>
     ::= <actual-parameter-list> ,
          <actual-parameter>

<actual-parameter>
     ::= <expression>
     ::= <array-identifier>
     ::= <procedure_ident if ier>

<safety-statement>
     ::= NOW-SAFE <id_list>
     ::= NOW-UNSAFE <id-list>
```

3.2 - SEMANTICS

ASSIGNMENT STATEMENTS

The assignment statement causes the value represented by an expression to be assigned to the variable appearing to the left of the assignment symbol You will see later (see page 22) that one value may be assigned to two or more variables through the use of two or more assignment symbols. The operation of the assignment statement proceeds in the following order

a) The subscript expressions of the left part variable (if any - Sail defines "variable" to include both array elements and simple variables) are evaluated from left to right (see Expression Evaluation Rules, page 23).

b) The expression is evaluated.

c) The value of the expression is assigned to the left part variable, with subscript expressions, if any, having values as determined in step a.

This ordering of operations may usually be disregarded. However it becomes important when expression assignments (page 22) or function calls with reference parameters appear anywhere in the statement. For example, in the statements:

```
I-3;
A[I]←3+(I←1);
```

A[3] will receive the value 4 using the above algorithm. A[1] will not change.

Any algebraic expression (REAL, INTEGER (BOOLEAN), or STRING) may be assigned to any variable of algebraic type. The resultant type will be that of the left part variable. The conversion rules for assignments involving mixed types are mildly amusing. They are identical to the conversion rules for combining mixed types in algebraic expresions (see page 2 1 below).

SWAP ASSIGNMENT

The ↔ operator causes the value of the variable on the left hand side to be exchanged with the value of the variable on the right hand side. Arithmetic (REAL↔INTEGER) type conversions are made, if necessary: any other type conversions are invalid. Note that the ↔ operator may not be used in assignment expressions.

CONDITIONAL STATEMENTS

These statements provide a means whereby the execution of a statement, or a series of statements, is dependent on the logical value produced by a Boolean expression.

A Boolean expression is an algebraic expression whose use implies that it is to be tested as a logical (truth) value. If the value of the expression is Ø or NULL, the expression is a FALSE boolean expression, otherwise it is TRUE. See about type conversion, page 21.

IF STATEMENT - The statement following the operator THEN (the "THEN part") is executed if the logical value of the Boolean expression is TRUE: otherwise, that statement is Ignored.

IF ELSE STATEMENT - If the Boolean expression is true. the "THEN part" is executed and the statement following the operator ELSE (the "ELSE part") is Ignored. If the Boolean expression is FALSE, the "ELSE part" is executed and the "THEN part" is ignored.

AMBIGUITY IN CONDITIONAL STATEMENTS
The syntax given here for conditional statements does not fully explain the correspondences between THEN-ELSE pairs when conditional statements are nested. An ELSE will be understood to match the immediately preceding unmatched THEN. Example:

```
COMMENT DECIDE WHETHER TO GO TO WORK;

IF -WEEKEND THEN
   IF GIANTS-ON-TV THEN BEGIN
      PHONE_EXCUSE("GRANDMOTHER DIED");
      ENJOY(GAME);
      SUFFER(CONSCIENCE_PANGS)
   END
   ELSE IF REALLY-SICK THEN BEGIN
      PHONE_EXCUSE("REALLY SICK");
      ENJOY(0);
      SUFFER( AGONY )
   END
   ELSE GO TO WORK;
```

GO TO STATEMENTS
Each of the three forms of the Go To statement means the same thing -- an unconditional transfer is to be made to the "target" statement labeled by the label Identifier. The following rules pertain to labels:

1) All label identifiers used in a program must be declared.

2) The declaration of a label must be local to the block immediately surrounding the statement it Identifies (see exception below). Note that compound statements (BEGIN-END pairs containing no declarations) are not blocks. Therefore the block

```
BEGIN "B1"
 INTEGER I,J; LABEL L1;
 ...
 IF BE3 THEN BEGIN "CI "
  ...
  L1: . . .
  ...
 END "CI ";
 ...
 GO TO L1
END "B1"
```

is legal.

3) Rule 2 can be violated if the inner block(s) have no array declarations. E.g.:

Legal	Illegal
BEGIN "B1"	BEGIN "B1"
INTEGER I,J;	INTEGER I,J;
LAEEL L1;	LABEL LI;
...	...
BEGIN "B2"	BEGIN "B2"
REAL X;	REAL ARRAY X [1:10];
...	...
L1: . . .	L1:. . .
...	...
END "B2";	END "B2";
GO TO L1;	GO TO L1;
END "B1"	END "B1"

4) No Go To statement may specify a transfer into a FOREACH statement (FOREACH statements are part of LEAP -- page 51), or into complicated For loops (those with For Lists or which contain a NEXT statement).

Labels will seldom be needed for debugging purposes. The block name feature (see page 96) and the listing feature which associates with each source line the octal address of its corresponding object code (see page 92) should provide enough information to find things easily.

Many program loops coded with labels can be alternatively expressed as For or While loops, augmented by DONE, NEXT, and CONTINUE statements. This often results in a source program whose organization is somewhat more transparent, and an object program which is more efficient.

FOR STATEMENTS
For, Do and While statements provide methods for forming loops in a program. They allow the repetitive execution of a statement zero or more times. These statements will be described by means of SAIL programs which are functionally equivalent but which demonstrate better the actual order of processing. Refer to these equations for any questions you might have about what gets evaluated when, and how many times each part is evaluated.

Let VBL be any algebraic variable, AE1, . . . , AE8 any algebraic expressions, BE a Boolean expression, TEMP a temporary location, S a statement. Then the following SAIL statements are equivalent:

Using For Statements --

```
FOR VBL ←AE1, AE2, AE3 STEP
   AE4 UNTIL AE5, AE6 STEP AE7 WHILE
   BE, AE8 DO S;
```

Equivalent formulation without For Statements --

```
     VBL←AE1;
     S;
     VBL←AE2;
     S;

     VBL-AE3; Comment STEP-UNTIL loop;
LOOP1: IF (VBL-AE5)*SIGN(AE4) ≤ 0 THEN
     BEGIN
        S;
        VBL-VBLtAE4;
        GO TO LOOP1
     END;

     VBL←AE6; Comment STEP-WHILE loop;
LOOP2: IF BE THEN BEGIN
        S;
        VBL←VBL+AE7;
        GO TO LOOP2
     END;

     VBL-AE8;
     S;
```

If AE4 (AE7) is an unsubscripted variable, changing its value within the loop will cause the new value to be used for the next iteration. If AE4 (AE7) is a constant or an expression requiring evaluation of some operator, the value used for the step element will remain constant throughout the execution of the For Statement. If AE5 is an expression, it will be re-evaluated before each iteration, so watch this possible source of ineff iciency.

Now consider the For Statement:

```
FOR VBL←AE1 STEP CONST UNTIL AE2 DO S;
```

where const is a positive constant. The compiler will simplify this case to:

```
     VBL←AE1;
LOOP3: IF VBL ≤ AE2 THEN BEGIN
        S;
        VBL←VBL+CONST;
        GO TO LOOP3
     END;
```

If CONST is negative, the line at LOOP3 would be:

```
LOOP3: IF VBL ≥ AE2 THEN BEGIN
```

The value of VBL when execution of the loop is terminated, whether it be by exhaustion of the For list or by execution of a DONE, NEXT or GO TO statement (see page 17, page 17, page 15), is the value last assigned to it using the algorithm above. This value is therefore always well-defined.

The statement S may contain assignment statements or procedure calls which change the value of VBL. Such a statement behaves the same way it would if inserted at the corresponding point in the equivalent loop described above.

WHILE STATEMENT

The statement:

```
WHILE BE DO S;
```

is equivalent to the statements:

```
LOOP: IF BE THEN BEGIN
        S;
        GO TO LOOP
      END;
```

DO STATEMENT

The statement:

```
DO S UNTIL BE;
```

is equivalent to the sequence:

```
LOOP: s;
      IF -BE THEN GO TO LOOP;
```

CASE STATEMENTS

The statement:

```
CASE AE OF BEGIN  SO; S1; S2 . . . Sn END
```

is functionally equivalent to the statements:

```
TEMP-AE;
IF TEMP<0 THEN ERROR
   ELSE IF TEMP = 0 THEN SO
   ELSE IF TEMP = 1 THEN S1
   ELSE IF TEMP = 2 THEN S2
   ...
   ELSE IF TEMP = n THEN Sn
   ELSE ERROR;
```

For applications of this type the CASE statement form will give significantly more efficient code than the equivalent If statements. Notice that dummy statements may be inserted for those cases which will not occur or for which no entries are necessary. For example,

```
CASE AE OF BEGIN SO; ; S3; ;; S6; END
```

provides for no actions when AE is 1,2,4,5, or 7. When AE is 0, 3, or 6 the corresponding statement will be executed. However, slightly more efficient code may be generated with a second type of Case statement that numbers each of its statement with [n] where n is an integer constant. The above example using this type of Case statement is then:

```
CASE AE OF BEGIN [3] S3; [0]S0;[6]S6 END;
```

All the statements must be numbered, and that the numbers must all be non-negative integers constant expressions, although them may be in any order.

Block names (i.e. any string constant) may be used after the BEGIN and END of a Case statement with the same effect as block names on blocks or compound statements. (see about block names on page 1).

RETURN STATEMENT

This statement is invalid if it appears outside a procedure declaration. It provides for an early return from a Procedure execution to the statement calling the Procedure. If no return statement is executed, the Procedure will return after the last statement representing the procedure body is executed (see page 7).

An untyped Procedure (see page 18) may not return a value. The return statement for this kind of Procedure consists merely of the word RETURN. If an argument is given, it will cause the compiler to issue an error message.

A typed Procedure (see page 24) must return a value as it executes a return statement. If no argument is present an error message will be given. If the Procedure has an algebraic type, any algebraic expression may be returned as its value; type conversion will be performed in a manner described on page 21.

If no RETURN statement is executed in a typed Procedure, the value returned is undefined (it could be anything -- try it, it's fun).

DONE STATEMENT

The statement containing only the word DONE may be used to terminate the execution of a FOR, WHILE, or DO (also FOREACH- see page 58) loop explicitly. Its operation can most easily be seen by means of an example. The statement

```
FOR I-1 STEP 1 UNTIL n DO BEGIN
   S;

   IF BE THEN DONE;
   ...
END
```

is equivalent to the statement

```
FOR I-1 STEP 1 UNTIL n DO BEGIN
   S;
   ...
   IF BE THEN GO TO EXIT;
   ...
 END;
EXIT:
```

In either case the value of I is well-defined after the statement has been executed (see page 16).

The DONE statement will only cause an escape from the innermost loop in which it appears, unless a block name follows "DONE". The block name must be the name of a block or compound statement (a "Loop Block") which is the object statement of some FOR, WHILE, or DO statement in which the current one is nested. The effect is to terminate all loops out to (and including) the Loop Block, continuing with the statement following this outermost loop. For example:

```
WHILE TRUE DO BEGIN "B1"
   ...
   IF OK THEN DO BEGIN "B2"
      ...
      FOR I←1 STEP 1 UNTIL K DO
         IF A[I]=FLAGWORD THEN DONE "B1";
      ...
      END "B2" UNTIL COWS-COME-HOME;
   ...
   END "B1";
```

Here the block named "B1" is the "loop block".

NEXT STATEMENT

A Next statement is valid only in a For Statement, While Statement, or Do Statement (or Foreach- see page 58). Processing of the loop statement is temporarily suspended. When the NEXT statement appears in a For loop, the next value is obtained from the For List and assigned to the controlled variable. The termination test is then made. If the termination condition is satisfied, control is passed to the statement following the For Statement. If not, control is returned to the inner statement following the NEXT statement. In While and Do loops, the termination condition is tested. If it is satisfied, execution of the loop terminates. Otherwise it resumes at the statement within the loop following the NEXT statement.

Unless a block name follows NEXT, the innermost loop containing the NEXT statement is used as the "Loop Block" (see page 17). The terminating condition for the loop block is checked. If the condition is met, all inner loops are terminated (in DONE fashion) as well. If continuation is indicated, no inner-loop FOR-variable or WHILE-condition will have been affected by the NEXT code.

The reserved word NEEDNEXT must precede FOR, WHILE, or DO in the "Loop Block", and must not appear between this block and the NEXT statement. Example:

```
NEEDNEXT WHILE ¬EOF DO BEGIN
   S←INPUT(1,1);
   NEXT;
   Comment check EOF and terminate if TRUE;
   T←INPUT(1,3);
      PROCESS_INPUT(S,T);
   END;
```

CONTINUE STATEMENT

The Continue statement is valid in only those contexts valid for the DONE statement (see page 17); the "Loop Block" is determined in the same way (i.e. implicitly or by specifying a block name). All loops out to the Loop Block are terminated as if DONE had been requested. Control is transferred to a point inside the loop containing the Loop Block, but after all statements in the loop. Example:

```
FOR I-i STEP 1 UNTIL N DO BEGIN
      ...
      CONTINUE;
      ...
END
```

is semantically equivalent to:

```
FOR I-1 STEP 1 UNTIL N DO BEGIN
      LABEL CONT;
      ...
      GO TO CONT;
      ...
 CONT:
END
```

PEOCEDURE STATEMENTS

A Procedure statement is used to invoke the execution of a Procedure (see page 7). After execution of the Procedure, control returns to the statement immediately following the Procedure statement. SAIL does allow you to use typed Procedures as procedure statements. The value returned from the Procedure is simply discarded.

The actual parameters supplied to a Procedure must in match the formal parameters described in the procedure declaration, modulo Sail type conversion. Thus one may supply an integer expression to a real formal, and type conversion will be performed as on page 21

If an actual parameter is passed by VALUE, only the value of the expression is given to the Procedure. This value may be changed or examined by the Procedure, but this will in no way affect any of the variables used to evaluate the actual parameters. Any algebraic expression may be passed by value. Neither Arrays nor Procedures may be passed by value (use ARRBLT, page 43, to copy arrays). See the default declarations for parameters in page 8.

If an actual parameter is passed by REFERENCE, its address is passed to the Procedure. All accesses to the value of the parameter made by the Procedure are made indirectly through this address. Therefore any change the Procedure makes in a reference parameter will change the value of the variable which was used as an actual parameter. This is sometimes useful. However if it is not Intended, use of this feature can also be sotnewhat confusing as well as moderately inefficient. Reference parameters should be used only where needed.

Variables, constants, Procedures, Arrays, and most expresslons may be passed by reference.No String expressions (or String constants) may be reference parameters.

If an expression is passed by reference, its value is first placed in a temporary location; a constant passed by reference is stored in a unique location. The address of this location is passed to the Procedure. Therefore, any values changed by the Procedure via reference parameters of this form will be inaccesible to the user after the Procedure call. If the called program is an assembly language routine which saves the parameter address, it is dangerous to pass expressions to it, since this address will be used by the compiler for other temporary purposes. A warning message will be printed when expressions are called by reference.

The type of each actual parameter passed by reference must match that of its corresponding formal parameter, modulo Sail type conversion. The exception is reference string formals, which must have string variables (of string array elements) actual passed to them. If an algebraic type mismatch occurs the compiler will create a temporary variable containing the converted value and pass the address of this temporary as the parameter. A warning message will be printed. An exception is made for Fortran calls (see page 19).

PROCEDURES AS ACTUAL PARAMENTERS

If an actual parameter to a Procedure PC is the name of a Procedure PR with no arguments, one of three things might happen:

1) If the corresponding formal parameter requires a value of a type matching that of PR (in the loose sense given above in page 18), the Procedure is evaluated and its value is sent to the Procedure PC.

2) If the formal parameter of PC requires a reference Procedure of identical type, the address of PR is passed to PC as the actual parameter.

3) If the formal parameter requires a reference variable, the Procedure is evaluated, Its result stored, and its address passed (as with expressions in the previous paragraph) as the parameter.

If a Procedure name followed by actual parameters appears as an actual parameter it is evaluated (see functions, page 24). Then if the corresponding formal

parameter requires a value, the result of this evaluation is passed as the actual parameter. If the formal parameter requires a reference to a value, it is called as a reference expression.

FORTRAN PROCEDURES
If the Procedure being called is a Fortran Procedure, all actual parameters must be of type INTEGER (BOOLEAN) or REAL. All such parameters are passed by reference, since Fortran will only accept that kind of call For convenience, any constant or expression used as an actual parameter to a Fortran Procedure is stored in a temporary cell whose address is given as the reference actual parameter.

It was explained in page 7 that formal parameters need not be described for Fortran Procedures. This allows a program to call a Fortran Procedure with varying nutnbers of arguments, a feature which exists in DEC Fortran. No type conversion will be performed for such parameters, of course. If type conversion is desired, the formal parameter declarations should be Included in the Fortran procedure declaration: SAIL will use them if they are present.

To pass an Array to Fortran, mention the address of its first element (e.g. A[Ø], or B[1,1]).

NOW-SAFE and NOW-UNSAFE
The NOW-SAFE and NOW-UNSAFE statements both take a list of Array names (names only - no indicies) foliowing them. From a NOW-SAFE until the end of the program or the next NOW-UNSAFE, the specified arrays will not have bounds checking code emitted for them. If an array has had a NOW-SAFE done on it, or has been declared SAFE, NOW-UNSAFE will cause bounds checking code to be emitted until the array is made safe again (if ever). Note that NOW-SAFE and NOW-UNSAFE are compile time statements. "IF BE THEN NOW-SAFE . .." will not work.

SECTION 4

ALGOL EXPRESSIONS

4.1 - SYNTAX

<expression>
 ::= <simple-expression>
 ::= <conditional_expression>
 ::= <assignment_expression>
 ::= <case-expression>

<conditional-expression>
 ::= IF <boolean_expression> THEN
 <expression> ELSE <expression>

<assignment-expression>
 ::= <variable> ← <expression>

<case-expression>
 ::= CASE <algebraic-expression> OF (
 <expression-list>)

<expression_list>
 ::= <expression>
 ::= <expression_list> , <expression>

<simple_expression>
 ::= <algebraic-expression>
 ::= <leap_expression>

<boolean-expression>
 ::= <expression>

<algebraic_expression>
 ::= <disjunctive_expression>
 ::= <algebraic_expression> ∨
 <disjunctive_expression>

<disjunctive_expression>
 ::= <negated_expression>
 ::= <disjunctive_expression> ∧
 <negated-expression>

<negated-expression>
 ::= ¬ <relational-expression>
 ::= <relational-expression>

<relational-expression>
 ::= <algebraic-relational>
 ::= <leap-relational>

<algebraic-relational>
 ::= <bounded-expression>
 ::= <relational-expression>
 <relational_operator>
 <bounded-expression>

<relational_operator>
 ::= <
 ::= >
 ::= =
 ::= ≤
 ::= ≥
 ::= ≠

<bounded-expression>
 ::= <adding-expression>
 ::= <bounded-expression> MAX
 <adding-expression>
 ::= <bounded-expression> MIN
 <adding_expression>

<adding-expression>
 ::= <term>
 ::= <adding-expression> <add-operator>
 <term>

<adding-operator>
 ::= +
 ::= -
 ::= LAND
 ::= LOR
 ::= EQV
 ::= XOR

<term>
 ::= <factor>
 ::= <term> <mult_operator> <factor>

<mult_operator>
::= *
::= /
::= %
::= LSH
::= ROT
::= MOD
::= DIV
::= &

<factor>
::= <primary>
::= <primary> T <primary>

<primary>
::= <algebraic-variable>
::= - <primary>
::= LNOT <primary>
::= ABS <primary>
::= <string-expression> [<substring_spec> 1
::= ∞
::= <constant> .
::= <function-designator>
::= LOCATION (<loc_specifier>)
::= (<algebraic-expression>)

<string-expression>
::= <algebraic-expression>

<substring_spec>
::= <algebraic-expression> TO
<algebraic-expression>
::= <algebraic-expression> FOR
<algebraic-expression>

<function-designator>
::= <procedure-call>

<loc_specifier>
::= <variable>
::= <array_ident if ier>
::= <procedure-identifier>
::= <label-identifier>

<algebraic_variable>
:= <variable>

4.2 - TYPE CONVERSION

Sail automatically converts between the data types Integer, Real, String and Boolean. The following table illustrates by description and example these conversions. The data type boolean is identical to integer under the mapping TRUE≠Ø and FALSE=Ø.

From \ To	INTEGER	REAL	STRING
INTEGER		Left Justify and raise to appropriate power. 1345→1.345@3 -678→-6.78@2	The right 7 bits are converted to a 1 character string with that ASCI I code. 4 8 → "Ø"
REAL	Drop decimal fractions. 1.345@2→134 -6.7999@1→-67 2.3@-2→ 0		Convert to integer then convert to string. 4.8@1 → "0" 4.899@1 → "Ø"
STRING	The ASCII code for the first character of string. "ØSUM"→ 4 8 NULL → 0	Convert to integer then to real. "ØSUM"→ 4.8~1 NULL → 0	

NOTES: The NULL string is converted to 0, but 0 is converted to the one character string with the ASCII code of 0. If the absolute value of an integer is greater than 134217728, then some low order significance will be lost in the conversion to real; otherwise, conversion to real and then back to integer will result in the same integer value. If a real number has magnitude greater than 134217728, then conversion to integer will produce an invalid result.

Conversion from real to integer can be sped by a factor of 8 if SHORT reals and integers are used. It is only necessary that one of the data types be SHORT: both the number to be converted and the variable need not be SHORT. SHORTness is a dominate quality in algebraic binary operations, That is, the sum of a SHORT real and a regular real will be treated as a SHORT real. SHORT integers and reals must have an absolute magnitude of less than 134217728.

The binary arithmetic, logical, and String operations which follow will accept combinations of arguments of any algebraic types. The type of the result of such an operation is sometimes dependent on the type of its arguments and sometimes fixed. An argument may be converted to a different algebraic type before the operation is performed. The following table describes the results of the arithmetic and logical operations given various combinations of Real and Integer inputs. ARG1 and ARG2 represent the types of the actual arguments, ARG1' and ARG2' represent the types of the arguments after any necessary conversions have been made.

OPERATION	ARG1	ARG2	ARG1'	ARG2'	RESULT
+ -	INT	INT	INT	INT	INT*
* ↑ %	REAL	INT	REAL	REAL	REAL
MAX MIN	INT	REAL	REAL	REAL	REAL
	REAL	REAL	REAL	REAL	REAL
LAND LOR	INT	INT	INT	INT	INT
EQV XOR	REAL	INT	REAL	INT	REAL
	INT	REAL	INT	REAL	INT
	REAL	REAL	REAL	REAL	REAL
LSH ROT	INT	INT	INT	INT	INT
	REAL	INT	REAL	INT	REAL
	INT	REAL	INT	INT	INT
	REAL	REAL	REAL	INT	REAL
/	INT	INT	REAL	REAL	REAL
	REAL	INT	REAL	REAL	REAL
	INT	REAL	REAL	REAL	REAL
	REAL	REAL	REAL	REAL	REAL
MOD DIV	INT	INT	INT	INT	INT
	REAL	INT	INT	INT	INT
	INT	REAL	INT	INT	INT
	REAL	REAL	INT	INT	INT

* If ARG2 is negative for the operatore "↑", then the result is real.

4.3 - SEMANTICS

CONDITIONAL EXPRESSIONS

A conditional expression returns one of two possible values depending on the logical truth value of the Boolean expression If the Boolean expression (BE) is true, the value of the conditional expression is the value of the expression following the delimiter THEN. If BE is false, the other value is used. If both expressions are of an algebraic type, the precise type of the entire conditional expression is that of the "THEN part".

Unlike the nested If statement problem, there can be no ambiguity for conditional expressions, since there is an ELSE part in every such expression. Example:

```
FOURTHDOWN(YARDSTOGO,YARDLINE,
    IF YARDLINE < 70 THEN PUNT ELSE
    IF YARDLINE < 90 THEN FIELDGOAL ELSE
    RUNFORIT)
```

ASSIGNMENT EXPRESSIONS

The somewhat weird syntax for an assignment expression (it is equivalent to that for an assignment statement) is nonetheless accurate: the two function Identically as far as the new value of the left part variable is concerned. The difference is that the value of this left part variable is also retained as the value of the entire expression. Assuming that the assignment itself is legal (following the rules given in page 14 above), the type of the expression is that of the left part variable. This variable may now participate in any surrounding expressions as if it had been given its new value in a separate statement on the previous line. Only the ← operator is valid in assignment expressions. The ↔ operator is valid only at statement level. Example:

```
IF (I←I+1)< 30 THEN I←0 ELSE I←I+1;
```

CASE EXPRESSIONS

The expressron

```
CASE AE OF (E0, El, E2, . . . , En)
```

is equivalent to:

```
IF AE=0 THEN EO
    ELSE IF AE=1 THEN El
    ELSE IF AE=2 THEN E2
    . . .
    ELSE IF AE-n THEN En
    ELSE ERROR
```

The type of the entire expression is therefore that of EO If any of the expressions E I En cannot be fit into this mold an error message is issued by the compiler. Case expressions differ from Case statements in that one may not use the [n] construct to number the expressions. Example:

```
OUT(TTY,CASE ERRNO OF("BAD DIRECTORY",
    "IMPROPER DATA MODE",
    "UNKNOWN I/O ERROR",
    . . .
    "COMPUTER IN BAD MOOD"));
```

SIMPLE EXPRESSIONS

Simple expressions are simple only in that they are not conditional, case, or assignment expressions. There are in fact some exciting complexities to be discussed with respect to simple expressions.

PRECEDENCE OF ALGEBRAIC OPERATORS

The binary operators in SAIL generally follow "normal" precedence rules. That is, exponentiations are performed before multiplications or divisions, which in turn are performed before additions and subtractions, etc. The bounding operators MAX and MIN are performed after these operations. The logical connectives ∧ and ∨, when they occur, are performed last (∧ before ∨). The order of operation can be changed by including parentheses at appropriate points.

In an expression where several operators of the same precedence occur at the same level, the operations are performed from left to right. See page 23 for special evaluation rules for logical connectives.

TABLE OF PRECEDENCE

↑
*/%& MOD DIV LSH ROT
+ - ⊗ ≡ LAND LOR
MAX MIN
= ≠ < ≤ > ≥
∧ ∨

EXPRESSION EVALUATION RULES

SAIL does not evaluate expressions in a strictly left-to-right fashion. If we are not constrained to a left-to-right evaluation, (as is ALGOL 6Ø), we can in some cases produce considerably better code than a strict left-to-right scheme could achieve. Intuitively, The essential features (and pitfalls) of this evaluation rule can be illustrated by a simple example:

```
b ← 2.6;
c ← b + (b ← b/2);
```

The second statement is executed as follows: divide b by 2 and assign this value (1.3) to b. Add this value to b and assign the sum to c. Thus c gets 2.6. If the expressions were evaluated in a strictly left-to-right manner, c would get 2.6 t 1.3.

The evaluation scheme can be stated quite simply: code is generated for the operation represented by a BNF production when the reduction of that BNF production takes place. That is, b t (b←b/2) isn't reduced until after (b←b/2) is reduced, so the smaller expression gets done first.

"∨" (OR)

If an algebraic expression has as its major connective the logical connective "∨", the expression has the logical value TRUE (arithmetic value some non-zero integer) if either of its conjuncts (the expressions surrounding the "∨") is true: FALSE otherwise. A∨B does NOT produce the bit-wise Or of A and B if they are algebraic expressions. Truth values combined by numeric operators will in general be meaningless (use the operators LOR and LAND for bit operations),

The user should be warned that in an expression containing logical connectives, only enough of the expression is evaluated (from left to right) to uniquely determine its truth value. Thus in the expression

```
(J<3 ∨ (K←K+1) > Ø),
```

K will not be incremented if J is less than 3 since the entire expression is already known to be true. Conversely in the expression

```
(X ≥Ø ∧ SQRT(X)>2)
```

there is never any danger of attempting to extract the square root of a negative X, since the failure of the first test testifies to the falsity of the entire expression -- the SQRT routine is not even called in this case.

"∧" (AND)

If a disjunctive expression has as its major connective the logical connective "∧", the expression has the logical value TRUE if both of its disjuncts are TRUE; FALSE otherwise. Again, if the first disjunct is FALSE a logical value of FALSE is obtained for the entire expression without further evaluation.

"¬" (NOT)

The unary Boolean operator ¬ applied to an argument BE(a relational expression, see Syntax) has the value TRUE if BE is false, and FALSE if BE is true. Notice that ¬A is not the bitwise complement of A, if A is an algebraic value. If used as an algebraic value, ¬A is simply 0 if A+0 and some non-zero Integer otherwise.

"<>≤≥=≠" (RELATIONS)

If any of the binary relational operators is encountered, code is produced to convert any String arguments to Integer numbers. Then type conversion is done as it is for the t operations (see page 21. The values thus obtained are compared for the indicated condition. A Boolean value TRUE or FALSE is returned as the value of the expression.. Of course, if this expression is used in subsequent arithmetic operations, a conversion to integer is performed to obtain an integer value.

MAX MIN

A MAX B (where A and B are appropriate expressions -- see the Syntax) has the value of the larger of A and B (in the algebraic sense). Type conversions are performed as if the operator were '+'. '0 MAX X MIN 10' is X if Ø≤X≤ 10, 0 if X<Ø, 10 if X>1Ø.

"+-" (ADDITION AND SUBTRACTION)

The t and - operators will do integer addition (subtraction) if both arguments are integers (or converted to integers from strings); otherwise, rounded Real addition or subtraction, after necessary conversions, is done.

LAND LOR XOR EQV LNOT

LAND, LOR, XOR, and EQV carry out bit-wise And, Or, Exclusive Or, and Equivalence operations on their arguments. No type conversions are done for these functions. The logical connectives ∧ and ∨ do not have this effect -- they simply cause tests and jumps to be compiled. The type of the result is that of the first operand. This allows expressions of the form X LAND '777777777, where X is Real, if they are really desired.

The unary operator LNOT produces the bitwise complement of its (algebraic) argument. No type conversions (except strings to integers) are performed on the argument. The type of the result (meaningful or not) is the type of the argument.

"*/%" (MULTIPLICATION AND DIVISION)
The operatron * (multiplication), like + and -, represents Integer multiplication only if both arguments are Integers; Real otherwise. Integer multiplication uses the IMUL tnachine Instruction -- no double-length result is available.

The / operator (division) always does rounded Real division, after converting any Integer arguments to Real.

The % operator has the same type table as +, -, and *. It performs whatever division is appropriate.

DIV MOD
DIV and MOD force both arguments to be integers before dividing. X MOD Y is the remainder after X DIV Y is performed:

X MOD Y = X - (X DIV Y)*Y .

LSH ROT
LSH and ROT provide logical shift operations on their first arguments. If the value of the second argument is positive, a shift or rotation of that many bits to the left is performed. If it is negative, a right-shift or rotate is done. To obtain an arithmetic shift (ASH) operation, multiply or divide by the appropriate power of 2. the compiler will change this operation to a shift operation

"&" (CONCATENATION)
This operator produces a result of type String. It is the String with length the sum of the lengths of its arguments, containing all the characters of the second string concatenated to the end of all the characters of the first. The operands will first be converted to strings if necessary as described in page 21 above. Numbers can be converted to strings representing their external forms (and vice-versa) through explicit calls on execution time routines like CVS and CVD (see page 3i below). NOTE: Concatenation of constant strings will be done at compile time where possible. For example, if SS is a string variable, SS&'12&'15 will resuit in two runtime concatenations, while SS&('12&'15) will result in one compile time concatenation and one runtime concatenation,

"↑" (EXPONENTIATION)
A factor is either a primary or a primary raised to a power represented by another primary. As usual, evaluation is from left to right, so that A↑B↑C is evaluated as (A↑B)↑C. In the factor X↑Y, a suitable number of multiplications and additions is performed to produce an "exact" answer if Y is a positive integer. Otherwise a routine is called to approximate ANTILOG(Y LOG X). The result has the type of X in the fortner case. It is always of type Real in the latter.

SUBSTRINGS
A String primary which is qualified by a substring specification represents a part of the specified string. ST[X FOR Y] represents the Xth through the (X + Y - 1)th characters of the String ST. ST[X TO Y] represents the Xth through Yth characters of ST.

Consider the ST[X TO Y] case. If Y > LENGTH(ST), (LENGTH is a runtime which returns the number of characters in the string -- see page 41) Y ← LENGTH(ST); if Y<Ø, Y←Ø; in either case the right half of the global Integer _SKIP_ is set to TRUE. If X ≤ 1 it is set to 1. If X > (the modified) Y, it is set to Y+1 (null string guaranteed). In either case the left half of _SKIP_ is made TRUE. The ST[X FOR Y] operation is converted to the ST[X TO Y] case before the substring operation is performed.

To examine the above conditions, declare EXTERNAL INTEGER _SKIP_, clear it, and look at it after any interesting substring operation.

"∞" (SPECIAL LENGTH OPERATOR)
This special primary construct is valid only within substring brackets. It is an algebraic value representing the length of the most immediate string under consideration. Example:

A[4 to ∞]	throws out the first 3 characters of A.
A[3 for B[∞- 1 for 1]]	uses the next to the last character of string B as the number of characters for the A substring operation.

FUNCTION DESIGNATORS
A function designator defines a single value. This value is produced by the execution of a typed user Procedure or of a typed execution-time routine (See chapters 7 and 9 for execution-time routines). For a function designator to be an algebraic primary, its Procedure must be declared to have an algebraic type Untyped Procedures may only be called from Procedure statements (see page 18). The value obtained from a user-defined Procedure is that provided by a Return Statement within that Procedure. If the Procedure does not execute a Return Statement, the value might be anything at all. A Return Statement in a typed Procedure must mention a value (see page 17)

The rules for supplying actual parameters in a function designator are identical to those for supplying parameters in a procedure statement (see page 18).

UNARY OPERATORS
The unary operator ABS is valid only for algebraic quantities It returns the absolute value of its argument

-X is equivalent to (0-X). No type conversions are performed.

MEMORY AND LOCATION
One's core image can be considered a giant one dimensional array, which may be accessed with the MEMORY construct.

```
MEMORY [ <integer expression> ]
```

One can store and retrieve from the elements of MEMORY just as with any other array. However, when retrieving from MEMORY, one can specify the type of the accessed element by including type declarator reserved words after the <integer expression>. For example:

```
. . . ← MEMORY[X, INTEGER]
. . . ← MEMORY[X, REAL]
. . . ← MEMORY[X, ITEMVAR]
COMMENT items and sets are part of Leap;
. . . ← MEMORY[X, SET]
. . . ← MEMORY[X, INTEGER ITEMVAR]
```

Note that one can not specify the contents of memory to be an array or a string.

LOCATION is a predeclared Sail routine that returns the index in MEMORY (i.e. the address in core relative to the starting address of one's program) of the Sail construct furnished it. The following is a list of constructs it can handle and what LOCATION will return.

CONSTRUCTx	LOCATION(x) RETURNS
variables	address of the variable
array name	address of a word containing the the address of the first word of the array header
array element	address of that element
procedure name	address of the procedures entry code
labels	address of the label

SECTION 5

ASSEMBLY LANGUAGE STATEMENTS

5 1 - SYNTAX

```
<code-block>
    ::= <code-head> <code_tail>

<code-head>
    ::= <code-begin>
    ::= <code-begin> <block-name>
    ::= <code-head> <declarat ion> ;

<code_begin>
    ::= START-CODE
    ::= QUICK-CODE

<code-tail>
    ::= <instruction> END
    ::= <instruction> END <block-name>
    ::= <Instruction> ; <code_tail>

<Instruction>
    := <addresses>
    ::= <opcode>
    ::= <opcode> <addresses>

<addresses>
    = <address>
    ::= <ac_field> ,
    ::= <ac_field> , <address>

<ac_field>
    = <constant-expression>

<address>
    ::= <indexed_address>
    ::= @ <Indexed-address>

<indexed_address>
    = <simple_address>
    = <simple_address> ( <index-field> )

<simple_address>
    ::= <identifier>
    ::= <constant-expression>
    ::= <literal>

<literal>
    ::= [ <constant-expression> I

<index-field>
    ::= <constant-expression>

<opcode>
    ::= <constant-expression>
    ::= PDP-_opcode
```

5.2 - SEMANTICS

Within a START-CODE (QUICK-CODE) block, statements are processed by a small and weak, but hopefully adequate, assembly language translator. Each "instruction" places one instruction word into the output file. An instruction consists of

```
<label>:<opcode>    <ac_field>,@<simple_addr> (<index>)
```

or some subset thereof (see syntax). Each instruction must be followed by a semi-colon.

DECLARATIONS IN CODE BLOCKS
A code-block behaves like any other block with respect to block structure. Therefore, all declarations are valid, and the names given in these declarations will be available only to the instructions in the code-block. All labels must be declared as usual. Labels in code-blocks may refer to instructions which will be executed, or to those which are not really instructions, but data to be manipulated by these instructions (these latter words must be bypassed in the code by jump instructions). The user may find it easier to declare variables or SAFE arrays as data areas rather than using labels and null statements. As noted below, identifiers of simple variables are addresses of core locations. Identif iers of arrays are addresses of the first word of the array header (see the appendix on array implementation).

PROTECT ACS DECLARATION

```
PROTECT-ACS <ac#>, . . . , <ac#>;
```

where <ac#> is an integer constant between 0 and ' 17, is a declaration. Its effect is to cause Sail not to use the named accumulators in the code it emits for

the block in which the declaration occurred (only AFTER the declaration). The most common use is with the ACCESS construct (see below); if one is using accumulators 2, 3, and 4 is a code block, then one should declare PROTECT-ACS 2,3,4 if one is going to use ACCESS This way, the code emitted by Sail for doing the ACCESS will not use accumulators 2, 3, or 4. WARNING: this does not prevent you from clobbering such ACs with procedure calls (your own procedures or Sail's). However, most Sail runtimes save their ACs and restore them after the call.

RESTRICTION: Accumulators P ('17), SP ('16), F ('12) and 1 are used for, respectively, the system PDL push down pointer, the string PDL push down pointer, the display pointer, and returning results from typed procedures and runtimes. More about these acs on page 27. The protect mechanism will not override these usages, so attempts to protect 1,'12,'16, or '17 will be futile.

OPCODES

The Opcode may be a constant provided by the user, or one of the standard (non I/O) PDP-10 operation codes, expressed symbolically. If a constant, it should take the form of a complete PDP-10 instruction, expressed in octal radix (e.g. DEFINE TTYUUO = "'5 10000000000";). Any bits appearing in fields other than the opcode field (first 9 bits) will be OR'ed with the bits supplied by other fields of instructions in which this opcode appears.

The Indirect, index, and AC fields have the same syntax and perform the same functions as they do in the FAIL or MACRO languages.

THE <simple addr> FIELD

1. If the <address> in an instruction is a constant (constant expression), it is assumed to be an Immediate or data operand, and is not relocated.

2. If the <address> is an identifier, the machine address (relative to the start of the compilation) is used, and will be relocated to the proper value by the Loader.

3. If the <address> is an identifier which has been declared as a formal parameter to a procedure, addressing arithmetic will be done automatically to get at the VALUE of the parameter. Hence if the <address> is a formal reference parameter, the instruction will be of the form OP AC,@-x('17) where x depends on exactly where the parameter is in the stack. If the formal was from a simple procedure, then '12 will be used as the index register rather than '17.

4. If a literal is used, the address of the compiled constant will be placed in the instruction.

5. Any reference to Strings will result in the address of the second descriptor word (byte pointer) to be placed in the instruction (see the appendix on string implementation for an explanation of string descriptors).

6. Accessing parameter of procedures global to the current procedure is difficult. ACCESS (<expr>) may be used to return the address of such parameters. ACCESS will in fact do all of the computing necessary to obtain the value of the expression <expr>, then return the address of that value (which might be a temporary). Thus, MOVE AC, ACCESS(GP) will put the value of the variable GP in AC, while MOVI AC, ACCESS(GP) will put the address of the variable GP in AC. If the expression is an item expression (see Leap), then the item's number will be stored in a temp, and that temp's address will be returned. The code emitted for an Access uses any acs that Sail believes are available, so one must include a PROTECT-ACS declaration in a Code block that uses ACCESS if you want to protect certain acs from being munged by the Access. WARNING: skipping over an Access won't do the right thing. For example,

```
SKIPE FLAG;
MOVE  '10, ACCESS ('777 LAND INTIN(CHAN));
MOVI  '16, 0;
```

will cause the program to skip into the middle of the code generated by the access if FLAG is 0.

START-CODE VERSUS QUICK-CODE

Before your instructions are parsed in a block starting with START-CODE, instructions are executed to leave all accumulators from 0 through '15 available for your use In this case, you may use a JRST to transfer control out of the code-block, as long as you do not leave (1) a procedure, (2) a block with array declarations, (3) a Foreach loop, (4) a loop with a For list, or (5) a loop which uses the NEXT construct. In a QUICK-CODE block, no accumulator-saving instructions are issued. Ac's '13 through '15 only are free. In addition, some recently used variables may be given the wrong values if used as address identifiers (their current values may be contained in Ac's Ø-'12); and control should not leave the code-block except by "falling through".

WARNING Concerning Default Radix: All integer constants will be expressed in decimal radix unless the octal representation is explicitly used.

ACCUMULATOR USAGE IN CODE BLOCKS

Although we have said that accumulators are "freed" for your use, this does not imply a complete carte

blanche Usually this means the compiler saves off values currently stored in the ACs which it wants to remember (the values of variables mostly), and notes that when the code block is finished, these ACs will have values in them that it doesn't care about. However, this is not the case with the following accumulators, which are not touched at all by the entrance and exit of code blocks:

NAME	NUMBER	USAGE
P	'17	The system push down list pointer. All procedures are called with a PUSHJ P, PROC and exited (usually) with a POPJ P. Use this as your PDL pointer in the code block, but be sure that its back to where it was on entrance to the block by the time you exit.
SP	'16	The string push down stack pointer. Used in all string operations. For how to do your own string mangling, see the implementation manual.
F	'12	This is used to maintain the "display" structure of procedures. DO NOT HARM AC F!! Disaster will result. A more exact description of its usage may be found in the appendix on procedures and the implemention manual.

CALLING PROCEDURES FROM INSIDE CODE BLOCKS
To call a procedure from inside a code block, say procedure PROT, say PUSHJ P, PROT. If the procedure requires parameters, PUSH P them in order before you PUSHJ P (i.e. the first one first, the second next, and the last right before the PUSHJ). If the formal is a reference, push the address of the actual onto the P stack If the formal is a value string, push onto the SP stack the two words of the string descriptor (see the appendix on string implementation for an explanation of string descriptors). If the formal is a reference string, simply PUSH P the address of the second word of the string descriptor (e.g. PUSH P, [S]). If the procedure is typed, it will return is value in AC 1 (a pointer to the second word if the procedure is a string procedure). More information can be found in the implementation manual and the appendix on procedure implementation.

NOTE: procedures will change your accumulators unless the procedure takes special pains to save and restore them

EXAMPLE:

```
INTEGER PROCEDURE PROT(REAL T; REFERENCE
        INTEGER TT; STRING TTT; REFERENCE
        STRING TTTT);
                BEGIN COMMENT BOOY: ENO:

DEFINE P = '17, SP = '16;

START-CODE
PUSH P . [3.14159];
PUSH P,[I];            COMMENT I is an integer variable;
MOVEI 1. S:            COMMENT Sis a string variable;
PUSH SP, -1 (1);       COMMENT if SAIL allowed address
                       arithmetic in Start-code, you
                       could have said PUSH SP, S-1:
PUSH SP. S;
PUSH P,[SS];           COMMENT SSis a string variable:
PUSHJ P,PROT;
END;
```

gives the same effect as

```
PROT(3.14159,I,S,SS);
```

SECTION 6

BACKTRACKING

6.1 - INTRODUCTION

Backup or backtracking is the ability to "back up" execution to a previous point. SAIL facilitiates backtracking by allowing one to REMEMBER, FORGET, or RESTORE variables in the new data type, CONTEXT.

6.2 - SYNTAX

```
<context-declaration>
    ::= CONTEXT <id-list>
    ::= CONTEXT ARRAY <array-list>
    ::= CONTEXT ITEM <id-list>
    ::= CONTEXT ITEMVAR <id-list>

<backtracking-statement>
    ::= <rem-keyword> <variable_list>
        <rem_preposition> <context-variable>

<rem-keyword>
    ::= REMEMBER
    ::= FORGET
    ::= RESTORE

<rem-preposition>
    ::= IN
    ::= FROM

<variable_list>
    ::= <vari_list>
    ::= ( <vari_list> )
    ::= ALL
    ::= <context-variable>

<vari_list>
    ::= <vari>
    ::= <vari_list> , <vari>

<vari>
    ::= <variable>
    ::= <array_identifier>

<context-variable>
    ::= <variable>

<array_identifier>
    ::= <identifier>

<context-element>
    ::= <context_variable> : <variable>
```

6.3 - SEMANTICS

THE CONTEXT DATA TYPE
A context is essentially a storage place of undefined capacity When we REMEMBER a variable in a context, we remember the name of the variable along with its current value (if an array, values). If we remember a value which we have already remembered in the named context, we destroy the old value we had remembered and replace it with the current value of the variable. Values can be given back to variables with the RESTORE statement.

Context variables are just like any other variables with respect to scope. Also, at execution time, context variables are destroyed when the block in which they were declared is exited in order to reclaim their space. Context arrays, items, and itemvars are legal (items and itemvars are part of Leap). NEW(<context variable>) is legal (NEW is also part of Leap).

RESTRICTIONS:

1. Context procedures do not exist. Use context itemvar procedures instead.
2. Context variables may only be passed by reference to procedures (i.e. contexts are not copied).
3. Contexts may not be declared "GLOBAL" (shared between jobs - Stanford only).
4. +,*,/, and all other arithmetic operators have no meaning when applied to Context variables. Therefore, context variable expressions always consist only of a context variable.

The empty context is NULL-CONTEXT. Context variables are Initialized to NULL-CONTEXT at program entry

REMEMBER

To save the current values of variables, list them, with or without surrounding parentheses, in the remember statement. All of an array will be remembered if subscripts of an array are not used, otherwise, only the value indicated will be remembered. If a variable has already been remembered in context, its value is replaced by the current value If one wants to update all the variables so far remembered in this context, one may say

```
REMEMBER ALL IN <context>.
```

If you have several contexts active,

```
REMEMBER CNTXTI IN CNTXT2
```

will note the variables Remembered in CNTXTI, and automatically Remember their CURRENT values in CNTXT2.

RESTORE

To restore the values of variables that were saved in a context, list them (with or without surrounding parentheses) in a restore statement. Restoring an array without using subscripts causes as much of the array that was remembered to be restored magically to the right locations in the array. You can remember a whole array, then restore all or selected parts (e.g. RESTORE A[1,2] FROM IX;). If you remembered only A[1,2], then restoring A will only update A[1,2]. RESTORE ALL IN IX will of course restore all the variables from IX. RESTORE CNTXT! FROM CNTXT2 will act like a list of the variables in CNTXTI was presented to the Restore instead of the identifier CNTXT 1.

Astute Leap user will have noted that the syntax for varrables includes Datum(typed itemvar) and similar things. If one executes REMEMBER DATUM (typed-item-expression-l) IN CNTXT, then RESTORE DATUM (<item_expression_2>) FROM CNTXT will give an error message unless the <typed_item_expression_2> returns the same item as <typed-item-expression-l>.

WARNING!!! Restoring variables that have been destroyed by block exits will give you garbage. For example, the following will blow up:

```
BEGIN "BLOWS UP"
   CONTEXT J1;
   INTEGER J;
   BEGIN INTEGER ARRAY L[1:J];
         REMEMBER J,L IN J1;
   END;
   RESTORE ALL FROM J1;
END "BLOWS UP";
```

FORGET

The forget statement just deletes the variable from the context without touching the current variable's value. Variables remembered in a context should be forgotten before the block in which the variables were declared is exited. FORGET ALL FROM XI and FORGET CNTXTI FROM CNTXT2 work just as the similar Restore statements work, only the variables are Forgotten instead of Restored.

IN-CONTEXT

The runtime boolean IN-CONTEXT returns true if the specified variable is in the specified context. For details, see page 43.

CONTEXT ELEMENTS

Context elements provide a convenient method of accessing a variable that is being remembered in a context. Examples of context elements:

```
CNTXT-VARI : SOME-VARI
DATUM(CNTXT_ITEM) : SOME-VARI
CNTXT_AR[2,3] : ARRY[4]
DATUM(CNTXT_VARI : ITMVR )
CNTXT-VARI :DATUM(ITMVR)
```

A context element is syntactically and semantically equivalent to a variable of the same type as the variable foliowing the colon. For the complete syntax of variables, see page 88. Assignments to context elements change the Remembered value (i.e. X←5; REMEMBER X IN C;C:X←6; RESTORE X FROM C; will leave X with the value 6).

As with the Restore statement, one may not use Context Elements of variables destroyed by block exits.

RESTRICTIONS: (1) One may not Remember Context Elements. (2)Passing Context Elements by reference to procedures that change contexts is dangerous. Namely, if the procedure Forgets the element that was passed to it by reference, then the user is left with a dangling pointer. A more subtle variation of this disaster occurs when the Context element passed is an array element. If the procedure Remembers the array that that array element was a part of, the formal that had the array element Context Element passed to it is left with a dangling pointer.

SECTION 7

INPUT/OUTPUT ROUTINES

7.1 - EXECUTION TIME ROUTINES IN GENERAL

SCOPE
A large set of pre-declared, built-in procedures and functions have been compiled into a library permanently resident on the system disk area (SYS:LIBSAn REL - n is the current version number), and optionally into a special sharable write-protected high segment. The library also contains programs for managing storage allocation and initialization, and for certain String functions. If a user calls one of these procedures, a request is automatically made to the loader to include the procedure, and any other routines it might need, in the core image (or to link to the high segment). These routines provide input/output (I/O) facilities, Arithmetic-String conversion facilities, array-handling procedures and miscellaneous other Interesting functions.

The remainder of this section and the next describes the calling sequences and functions of these routines.

NOTATIONAL CONVENTIONS
A short-hand is used in these descriptions for specifying the types (if any) of the execution-time routines and of their parameters. Before the description of each routine there is a sample call of the form

```
VALUE ← FUNCTION (ARG1, ARG2, . . . ARGn)
```

if VALUE is omitted, the procedure is an untyped one, and may only be called at statement level (page 18).

The types of VALUE and the arguments may be determined using the following scheme:

1) If " characters surround the sample identifier (which is usually mnemonic in nature) a String argument is expected. Otherwise the argument is Integer or Real. If it is important which of the types Integer or Real must be presented, it will be made clear in the description of the function. Otherwise the compiler assumes Integer arguments (for those functions which are predeclared). The user may pass Real arguments to these routines by re-declaring them in the blocks in which the Real arguments are desired.

2) If the @ character precedes the sample identifier, the argument will be called by reference. Otherwise it is a value parameter.

Example:

```
"RESULT" ← SCAN (@"SOURCE", BREAK-TABLE, @BRCHAR)
```

is a predeclared procedure with the implicit declaration:

```
EXTERNAL STRING PROCEDURE SCAN
    (REFERENCE STRING SOURCE;
     INTEGER BREAK-TABLE;
     REFERENCE INTEGER BRCHAR);
```

7.2 - I/O CHANNELS AND FILES

OPEN

```
OPEN(CHANNEL, "DEVICE", MODE,
    NUMBER-OF-INPUT-BUFFERS,
    NUMBER-OF-OUTPUT-BUFFERS,
    @COUNT, @BRCHAR, @EOF);
```

SAIL input/output operates at a very low level in the following sense: the operations necessary to obtain devices, open and close files, etc. , are almost directly analogous to the system calls used in assembly language. OPEN is used to associate a channel number (Ø to '17) with a device, to determine the data mode of the I/O to occur on this channel (character mode, binary mode, dump mode, etc.), to specify storage requirements for the data buffers used in the operations, and to provide the system with information to be used for input operations.

CHANNEL is a user-provided channel number which will be used in subsequent I/O operations to identify the device. CHANNEL may range from 0 to 15 ('17). If some file is already open on this channel,' a RELEASE will be performed for that channel before the OPEN is executed.

DEVICE must be a String (i.e. "TTY", "DSK") which is recognizable by the system as a physical or logical device name.

MODE is the data mode for the I/O operation. MODE 0 will always work for characters (see page 35 and page 36). Modes 8 ('1Ø) and 15 ('17) are applicable for binary and dump-mode operations using the functions WORDIN, WORDOUT, ARRYIN, or ARRYOUT (see page 36 and following). For other data modes, see

[Moorer]. If any of bits 18-21 are on in the MODE word, the I-O routines will not print error messages when data errors occur which present the corresponding bits as a response to the GETSTS UUO. Instead, the GETSTS bits will be reported to the user as described under EOF below. If bit 23 is on, no error message will be printed if an invalid file name specification is presented to LOOKUP, ENTER, or RENAME, a code identifying the problem will be returned (see page 33 and following, page 33 for details). If you don't understand any of this, leave all non-mode bits off in the MODE word.

NUMBER-OF-I INPUT/OUTPUT}_BUFFERS specifies the number of buffers to be reserved for the I/O operations (see [Moorer] for details). At least one buffer must be specified for input if any input is to be done in modes other than '17; similarly for output. If data is only going one direction, the other buffer specification should be 0. Two buffers give reasonable performance for most devices-i 1 is sufficient for a TTY, more are required for DSK if rapid operation is desired). The left half of the BUFFER parameter, if non-zero, specifies the buffer size for the I/O buffers. Use this only if you desire non-standard sizes.

The remaining arguments are applicable only for INPUT (String input). They will be ignored for any other operations (although their values may be changed by the Open function).

COUNT designates a variable which will contain the maximum number of characters to be read from "DEVICE" in a given INPUT call (see page 35, page 33). Fewer characters may be read if a break character is encountered or if an end of file is detected. The count should be a variable or constant (not an expression), since its address is stored, and the temporary storage for an expression may be re-used.

BRCHAR designates a variable into which the break character (see INPUT and BREAKSET again) will be stored. This variable can be tested to determine which of many possible characters terminated the read operation.

EOF designates a variable to be used for two purposes:

1) If EOF is 0 when OPEN is called, a SAIL error message will be invoked if the device is not available. The user will be given the options of retrying or terminating the operation. If EOF is non-zero when OPEN is called, it will be set to 0 if the OPEN is successful. Otherwise it will not be changed. In this case (EOF non-zero on entry) control will be returned to the user. This flag may then be tested.

2) EOF will be made non-zero (TRUE! if an end of file condition, or any error condition among those enabled (see MODE, above) is detected during any SAIL input/output operation. It will be 0 (FALSE) on return to the user otherwise. Subsequent inputs after an EOF return will return non-zero values in EOF and a null String result for INPUT. For ARRYIN , a 0 is returned as the value of the call after end of file is detected. If EOF is TRUE after such an operation, it will contain the entire set (18 bits) of GETSTS information in the left half. The EOF bit is '20000, and is the only one you'll ever see if you haven't specially enabled for others. A summary of the enable bits, the EOF and error bits, and their meanings is contained in the Appendix on page 99.

Assembly Language Approximation to OPEN:

```
        INIT    CHANNEL,MODE
        SIXBIT  /DEVICE/
        XWD     OHED,IHED
        JRST    <handle error condition>
        JUMPE   <NUMBER_OF_OUTPUT_BUFFERS>,GETIN
        <allocat buffer space>
        OUTBUF  CHANNEL,NUMBER_OF_OUTPUT_BUFFERS
GETIN:  JUMPE   <NUMBER_OF_INPUT_BUFFERS>,DONE
        <allocate buffer space>
        INBUF   CHANNEL,  NUMBER-OF-INPUT-BUFFERS
DONE:   <mark channel open -- internal bookkeeping>
        <return>

OHED:   BLOCK 3
IHED:   BLOCK 3
```

CLOSE, CLOSIN, CLOSO

CLOSE (CHANNEL)
CLOSIN (CHANNEL)
CLOSO (CHANNEL)

The input (CLOSIN) or output(CLOSO) side of the specified channel is closed: all output is forced out (CLOSO): the current fire name is forgotten. However the device is still active; no OPEN need be done again before the next input/output operation. Always CLOSE output files: SAIL exit code will deassign the device,

but does not force out any remaining output; you must do a CLOSE when writing on a disk file to have the new file (or a newly edited old file) entered on your User File Directory. No INPUT, OUT, etc. may be given to a directory device until an ENTER, LOOKUP, or RENAME has been issued for the channel.

CLOSE is equivalent to the execution of both CLOSIN and CLOSO for the channel.

GETCHAN

VALUE ← GETCHAN

The number of some channel not currently open is returned. -1 is returned if all channels are busy.

RELEASE

RELEASE (CHANNEL)

If an OPEN has been executed for this channel, a CLOSE is now executed for it. The device is dissociated from the channel and returned to the resource pool (unless it has been assigned by the monitor ASSIGN command). No I/O operation may refer to this channel until another OPEN denoting it has been executed.

Release is always valid. If the channel mentioned is not currently open, the command is simply ignored.

LOOKUP, ENTER

LOOKUP (CHANNEL , "FILE" , @FLAG);
ENTER (CHANNEL , "FILE" , @FLAG);

Before input or output operations may be performed for a directory device (DECtape or DSK) a file name must be associated with the channel on which the device has been opened (see page 31). LOOKUP names a file which is to be read. ENTER names a file which is to be created or extended (see [Moorer]). Both operations are valid even if no filename is really necessary. It is recommended that an ENTER be performed after every OPEN of an output device so that output not normally directed to the DSK can be directed there for later processing if desired. The format for a file name string is

"NAME", or
"NAME. EXT", or
"NAME[P,PN]", or
"NAME. EXT[P,PN]"

See [Moorer] for the meaning of these things if you do not immediately understand.

SAIL is not as choosy about the characters it allows as PIP and other processors are. Any character which is not ".", ",", "[", or "]" will be passed on. Up to 6 characters from NAME, 3 from EXT, P, or PN will be used -- the rest are ignored.

If the LOOKUP or ENTER operation fails (see [Moorer]) then variable FLAG may be examined to determine the cause. The left half of FLAG will be set to '777 777 (Flag has the logical value TRUE). The right half will contain the code returned by the system giving the cause of the failure. An invalid file specification will return a code of '10. In this case, if the appropriate bit (bit 23, see OPEN) was OFF in the MODE parameter of the OPEN, an error message will be printed; otherwise, the routine just returns without performing the UUO.

If the LOOKUP or ENTER succeeds, FLAG will be set to zero (FALSE).

RENAME

RENAME (CHANNEL , "FILE-SPEC" ,
PROTECTION , @FLAG);

The file open on CHANNEL is renamed to FILE-SPEC (a NULL file-name will delete the file) with read/write protection as specified in PROTECTION (nine bits, described in [Moorer]). FLAG is set as in LOOKUP and ENTER.

7.3 - BREAK CHARACTERS

BREAKSET

BREAKSET(TABLE, "BREAK-CHARS" , MODE);

Character input/output is done using the String features of SAIL. In fact, I/O is the chief justification for the existence of strings in the language.

String input presents a problem not present in String output. The length of an output String can be used to determine the number of characters written. However it is often awkward to require an absolute count for input. Quite often one would like to terminate input, or "break", when one of a specified set of characters is encountered in the input stream. In SAIL, this capability is implemented by means of the BREAKSET,

INPUT, TTYIN, and SCAN functions. The value of TABLE may range from 1 to 18. Thus up to 18 different sets of break specifications may exist at once. Which set will be used is determined by the TABLE parameter in an INPUT or SCAN function call. The function of a given BREAKSET command depends on the MODE, an integer which is interpreted as a right-justified ASCII character whose value is intended to be vaguely mnemonic. BREAKSET commands can be partitioned into 3 groups according to mode:

GROUP 1 -- Break character specifications

MODE FUNCTION

"I" (by Inclusion) The characters in the BREAK-CHARS String comprise the set of characters which will terminate an INPUT (or SCAN).

"X" (by eXclusion) Only those characters (of the possible 128 ASCII characters) which are NOT contained in the String BREAK-CHARS will terminate an input when using this table.

"O" (Omit) The characters in "BREAK-CHARS" will be omitted (deleted) from the input string.

Any "I" or "X" command completely specifies the break character set for its table (i.e., the table is reset before these characters are stored in it). Neither will destroy the omitted character set currently specified for this table. Any "O" command completely specifies the set of omitted characters, without altering the break characters for the table in question. If a character is a break-character, any role it might play as an omitted character is sacrificed.

The second group of MODEs determines the disposition of break characters in the input stream. The "BREAK-CHARS" argument is ignored in these commands, and may in fact be NULL:

GROUP 2 -- Break character disposition

MODE FUNCTION

"S" (Skip -- default mode) After execution of an "S" command the break character will not appear either in the resultant String or in subsequent INPUTs or SCANs-- the character is "skipped". Its value may be determined after the INPUT by examination of the break character variable (see page 31).

"A" (Append) The break character (if there is one -- see page 31 and page 35) is appended, or concatenated to the end of the Input string It will not appear again in subsequent inputs.

"R" (Retain) The break character does not appear in the resultant INPUT or SCAN String, but will be the first character processed in the next operation referring to this input source (file or SCAN String).

For disk and tape fiies using the standard editor format, line numbers present a special problem. A line number is a word containing 5 ASCII characters representing the number in bits 0-34, with a "1" in bit 35 No other words in the file contain l's in bit 35. Since String manipulations provide no way for distinguishing line numbers from other characters, there must be a way to warn the user that line numbers are present, or to allow him to ignore them entirely.

The third group of MODEs determines the disposition of these line numbers. Again, the "BREAK-CHARS" argument is ignored:

Group 3 -- Line number disposition

MODE FUNCTION

"P" (Pass -- default) Line numbers are treated as any other characters. Their identity is lost; they simply appear in the result string.

"N" (No numbers) No line number (or the TAB which always follows it in standard files) will appear in the result string. They are simply discarded.

"L" (Line no. break) The result String will be terminated early if a line number is encountered. The characters comprising the line number and the associated TAB will appear as the next 6 characters read or scanned from this character source. The user's break character variable (see page 31 and page 35) will be set to -1 to indicate a line number break.

"E" (lee Erman's very own mode) The result String is terminated on a line number as with "L", but neither the line number nor the TAB following it will appear in subsequent inputs. The line number word, negated, is returned in the user's (integer) BRCHAR variable.

"D" (Display) If the TTY is a DPY, each line number from any input file will be displayed (along with a page number) on the rrght-hand side of the screen. This mode really applies to all input operations after the "D" operand appears in any Breakset call. There is no way to turn it off.

Once a break table is set up, it may be referenced in an INPUT, TTYIN or SCAN call to control the scanning operation.

Example:

To delitnit a "word", a program might wish to input characters until a blank, a TAB, a line feed, a comma, or a semicolon is encountered, ignoring line numbers. Assume also that carriage returns are to be ignored, and that the break character is to be retained in the character source for the next scanning operation:

```
BREAKSET(DELIMS," ,;"&TAB&LF,"I");
    Comment break on any of these;

BREAKSET(DELIMS,'15,"O");
    Comment ignore carriage return;

BREAKSET(DELIMS,NULL,"N");
    Comment ignore line numbers;

BREAKSET(DELIMS,NULL,"R");
    Comment save break char for next time;
```

SETBREAK

SETBREAK(TABLE , "BREAK-CHARS" ,
"OMIT-CHARS" , "MODES")

SETBREAK is logically equivalent to the SAIL statement:

```
BEGIN "SETBREAK"
 INTEGER I;

  IF LENGTH(OMIT_CHARS) > 0 THEN
    BREAKSET(TABLE,OMIT_CHARS,"O");

  FOR I-1 STEP 1 UNTIL LENGTH(MODES) DO
    BREAKSET(TABLE,BREAK_CHARS,MODES[I FOR 1])

END "SETBREAK"
```

STDBRK

. STDBRK (CHANNEL);

Eighteen breakset tables have been selected as representative of the more common input scanning operations. The function STDBRK initializes the breakset tables by opening the file SYS:BKTBL.BKT on CHANNEL and reading in these tables. The user may then reset those tables which he does not like to something he does like.

The eighteen tables are described here by giving the SETBREAKs which would be required for the user to initialize them:

```
DELIMS ← '15 & '12 & '40 & '11 & '14;
Comment carriage return, line feed, space,
        tab, form feed;
LETTS ← "ABC . . . Zabc . . . z_";
DIGS ← "0123456789";
SAILID ← LETTS&DIGS;

SETBREAK(1' 12 ', 15 ',"INS" );
SETBREAK( 2; '12, NULL, "INA" );
SETBREAK( 3, DELIMS, NULL, "XNR" );
SETBREAK( 4, SAILID, NULL, "INS" );
SETBREAK( 5, SAILID, NULL, "INR" );
SETBREAK( 6, LETTS, NULL, "XNR" );
SETBREAK( 7, DIGS, NULL, "XNR" );
SETBREAK( 8, DIGS, NULL, "INS" );
SETBREAK( 9, DIGS, NULL, "INR" );
SETBREAK(10, DIGS&"+-@. ", NULL, "XNR" );
SETBREAK (11, DIGS&"+-@.", NULL, "INS" );
SETBREAK (12, DIGS&"+-@.", NULL, "INR" );
SETBREAK (13-18, NULL, NULL, NULL );
```

7.4 - I/O ROUTINES

INPUT

"RESULT" ← INPUT(CHANNEL, BREAK-TABLE);

A string of characters is obtained for the file open on CHANNEL, and is returned as the result. The INPUT operation is controlled by BREAK-TABLE (see page 33) and the reference variables BRCHAR, EOF, and COUNT which are provided by the user in the OPEN function for this channel (see page 31). Input may be terminated in several ways. The exact reason for termination can be obtained by examining BRCHAR and EOF:

EOF	BRCHAR	
≠Ø	0	End of file or an error (if enabled, see page 31) occurred while reading. The result is a String containing all non-omitted characters which remained in the file when INPUT was called.
0	0	No break characters were encountered. The result is a String of length equal to the current COUNT specifications for the CHANNEL (see page 31).
0	<Ø	A line number was encountered and the break table specified that someone wanted to know. The result String contains all characters up to the iine number. If mode "L" was specified in

the Breakset setting up this table, bit 35 is turned off in the line number word so that it will be input next time. -1 is placed in BRCHAR. If mode "E" was specified, the line number will not appear in the next input String, but its negated ASCII value, complete with low-order line number bit, will be found in BRCHAR.

0 >Ø A break character was encountered. The break character is stored in BRCHAR (an INTEGER reference variable, see page 31) as a right-justified 7-bit ASCII value. It may also be tacked on to the end of the result String or saved for next time, depending on the BREAKSET mode (see page 33).

If break table 0 is specified, the only criteria for termination are end of file or COUNT exhaustion. The routine is somewhat faster operating in this mode.

SCAN

"RESULT" ← SCAN (@"SOURCE" ,
BREAK-TABLE , @BRCHAR)

SCAN functions identically to INPUT with the following exceptions:

1. The source is not a data file but the String SOURCE, called by reference. The String SOURCE is truncated from the left to produce the same effect as one would obtain if SOURCE were a data file. The disposition of the break character is the same as it is for INPUT.

2. BRCHAR is directly specified as a parameter. INPUT gets its break character variable from a table set up by page 31,

3 Line number considerations are irrelevant.

OUT

OUT(CHANNEL,"STRING")

STRING is output to the file open on CHANNEL. If the device is a TTY, the String will be typed immediately. Buffered mode text output is employed for this operation. The data mode specified in the OPEN for this channel must be 0 or 1. The EOF variable will be set non-zero as described in page 31 if an error is detected and the program is enabled for it; 0 otherwise

LINOUT

LINOUT (CHANNEL , NUMBER);

ABS(NUMBER) mod 100,000 is converted to a 5 character ASCII string. These characters are placed in a single word in the output file designated by CHANNEL with the low-order bit (line-number bit) turned on. A tab is inserted after the line number. Mode 0 or 1 must have been specified in the OPEN (page 31) for the results to be anywhere near satisfactory. EOF is set as in OUT.

WORDIN

VALUE ← WORDIN (CHANNEL)

The next word from the file open on CHANNEL is returned. A 0 is returned, and EOF (see page 31, page 35) set, when end of file or error is encountered. This operation is performed in buffered mode or dump mode, depending on the mode specification in the OPEN. See the warning about Dump Mode IO, page 37.

ARRYIN

ARRYIN (CHANNEL , @LOC , HOW-MANY);

HOW-MANY words are read from the device and file open on CHANNEL, and deposited in memory starting at location LOC. Buffered-mode input is done if MODE (see page 31) is '10 or '14. Dump-mode input is done if MODE is '16 or '17. Other modes are illegal See the warning about Dump Mode IO, page 37. If an end of file or enabled error condition occurs before HOW-MANY words are read, the EOF variable (see page 31) is set to the enabled bits in its left half, as usual. Its right half contains the number of words actually read. EOF will be 0 if the full request is satisfied.

WORDOUT

WORDOUT (CHANNEL , VALUE);

VALUE is placed in the output buffer for CHANNEL. An OUTPUT is done when the buffer is full or when a CLOSE or RELEASE is executed for this channel. Dump

mode output will be done if dump mode is specified in the OPEN (see page 3 1). EOF is set as in OUT. See the warning about Dump Mode IO, page 37.

ARRYOUT

ARRYOUT (CHANNEL , @LOC , HOW-MANY);

HOW-MANY words are written from memory, starting at location LOC, onto the device and file open on channel CHANNEL. The valid modes are again '10, '14, '16, and '17. The EOF variable is set as in ARRYIN, except that the EOF bit itself will never occur,

WARNING ABOUT DUMP MODE IO
Any Dutnp Mode (modes '15 thorough '17) input which does not specify an n*128-word count will have the effect of losing the words up to the next 128-word boundry -- you'll get the next word(s) of the next 128-word record on the next input. Similarly:, any Dump Mode output fills out the file with 0's until a 128-word boundry is reached. Therefore, Dump Mode IO is not practical for sizes other than 128-word transfer multiples, in general.

MTAPE

MTAPE (CHANNEL , MODE);

MTAPE is Ignored unless the device associated with CHANNEL is a magnetic tape drive. It performs tape actions as follows:

MODE	FUNCTION
"A"	Advance past one tape mark (or file)
"B"	Backspace past one tape mark
"E"	Write tape mark
"F"	Advance one record
"R"	Backspace one record
"S"	Write 3 inches of blank tape
"T"	Advance to logical end of tape
"U"	Rewind and unload
"W"	Rewind tape

USETI, USETO

USETI (CHANNEL , VALUE);
USETO (CHANNEL , VALUE);

The corresponding system function is carried out (see [Moorer I).

REALIN, INTIN

VALUE ←REALIN(CHANNEL);
VALUE ←INTIN(CHANNEL);

Number input may be obtained using the functions REALIN or INTIN, depending on whether a Real number or an Integer is required. Both functions use the same free field scanner, and take as argument a channel number.

Free field scanning works as follows: characters are scanned one at a time from the input channel. Nulls, line numbers,and carriage returns are ignored. When a digit is scanned it is assumed that this is a number and the following syntax is used.

<number>
    ::= <sign> <real number>

<real number>
    ::= <decimal number>
    ::= <decimal number> <exponent>
    ::= <exponent>

<decimal number>
    ::= <integer>
    ::= <integer> .
    ::= <integer> . <integer>
    ::= . <integer>

<integer>
    ::= <digit>
    ::= <integer> <digit>

<exponent>
    ::= @<sign> <integer>

<sign>
    ::= t
    ::= -
    = <empty>

If the digit is not part of a number an error message will be printed and the program will halt. Typing a carriage return will cause the input function to return zero.

On input, leading zeros are ignored. The ten most significant digits are used to form the number. A check for overflow and underflow is made and an error

message printed if this occurs. When using INTIN any exponent is removed by scaling the Integer number. Rounding is used in this process. All numbers are accurate to one half of the least significant bit.

After scanning the number the last delimiter is replaced on the input string and is returned as the break character for the channel. If no number is found, a zero is returned, and the break variable is set to - 1; If an end of file or enabled error is sensed this is also returned in the appropriate channel variable. The maximum character count appearing in the OPEN call is ignored.

REALSCAN, INTSCAN

```
VALUE ← REALSCAN ( @"NUMBER_STRING" ,
                   @BRCHAR ) ;
VALUE ← INTSCAN ( @"NUMBER_STRING" ,
                   @BRCHAR );
```

These functions are identical in function to REALIN and INTIN Their inputs, however, are obtained from their NUMBER_STRING arguments. These routines replace NUMBER-STRING by a string containing all characters left over after the number has been removed from the front

7.5 - TELETYPE AND PSEUDO-TELETYPE ROUTINES

TELETYPE I/O ROUTINES

```
CHAR ← INCHRW:
CHAR ← INCHRS:
"STR" ← INCHWL.
"STR" ← INCHSL (@FLAG);
"STR" ← INSTR ( BRCHAR );
"STR" ← INSTRL ( BRCHAR );
"STR" ← INSTRS (@FLAG , BRCHAR );
"STR" ← TTYIN( TABLE , @BRCHAR );
"SIR" ← TTYINL ( TABLE , @BRCHAR);
"STR" ← TTYINS ( TABLE , @BRCHAR );
OUTCHR ( CHAR );
OUTSTR ( "STR" );
CLRBUF;
BACKUP;
LODED ( "STR" );
```

Each of the I/O functions uses the TTCALL UUO's to do direct TTY I/O.

INCHRW waits for a character to be typed and returns that character.

INCHRS returns a negative value if no characters have been typed: otherwise it is INCHRW.

INCHWL waits for a line, terminated by a carriage-return and line feed (CR-LF) to be typed. It returns as a string all characters up to (not including) the CR. The LF is lost, The line may also be terminated by any control (or meta at Stanford) character: the character will be included in the string result.

INCHSL returns NULL with FLAG ≠ 0 if no lines have been typed. Otherwise it sets FLAG to 0 and performs INCHWL.

INSTR returns as a string all characters up to, but not Including, the first instance of BRCHAR The BRCHAR instance is lost.

INSTRL waits for a line to be typed, then performs INSTR.

INSTRS is INCHSL if no lines are waiting: INSTRL otherwise.

TTY IN uses the break table features described in page 33 and page 35 to return a string and break character. Mode "R" is illegal: line number modes are irrelevant. The input count (see page 31) is set at 100.

TTYINL waits for a line to be typed, then does TTY IN.

TTYINS sets ERCHAR to 20 and returns NULL if no lines are waiting. Otherwise it is TTYINL.

OUTCHR types its character argument (right-justified in an integer variable).

OUTSTR types its string argument until the end of the string or a null character is reached.

CLRBUF flushes the input buffer.

BACKUP backs up the scan (when started by a system command).

LODED loads the line editor with the string argument.

PSEUDO-TELETYPE FUNCTIONS

```
LINE ← PTYGET ,
PTYREL ( LINE );
CHARACTERISTICS ← PTYGTL ( LINE );
```

PTYSTL (LINE , CHARACTERISTICS
NUMBER ← PTIFRE (LINE);
NUMBER ← PTOCNT (LINE);
CHAR ← PTCHRW (LINE);
CHAR ← PTCHRS (LINE);
PTOCHS (LINE , CHAR);
PTOCHW (LINE , CHAR);
PTOSTR (LINR , "STR");
"STR" ← PTYALL (LINE);
"STR" ← PTYSTR (LINE , BRCHAR);
"STR" ← PTYIN (LINE, BKTBL , @BRCHAR);

Pseudo-teletype functions are available at Stanford only.

PTYGET gets a new pseudo-teletype line number and returns it. The global variable _SKIP_ is -1 if the attempt to get a PTY was successful, and 0 otherwise.

PTYREL releases PTY identified by "line".

PTYGTL returns line characteristics for the PTY.

PTYSTL sets line characteristics for the PTY specified by "line".

PTIFRE returns the number of free characters in the PTY input buffer.

PTOCNT returns the number of free characters in the PTY output buffer.

PTCHRW waits for a character from the PTY and returns it.

PTCHRS reads a character from the PTY if there is one, returns -1 if none.

PTOCHS tries to send a character to a PTY. If the attempt was successful, the global variable _SKIP_ is -1, otherwise 0.

PTOCHW sends a character to a PTY, waiting if necessary.

PTOSTR sends the string to the PTY, waiting if necessary.

PTYALL returns whatever is in the PTY's output buffer. No warting is done.

PTYSTR reads characters from the PTY, waiting if necessary, until a character equal to "char" is seen. All but the break character is returned as the string. If the break character was '15 (carriage return), the following line-feed is snarfed.

PTY IN reads from the PTY (waiting -if necessary) according to break table convent ions. The break character is stored in "brchar".

SECTION 8

EXECUTION TIME ROUTINES

Please read Execution Time Routines in General, page 31, if you are unfamiliar with the format we use to describe runtime routines.

8.1 - TYPE CONVERSION ROUTINES

SETFORMAT

SETFORMAT (WIDTH , DIGITS);

This function allows specification of a minimum width for strings created by the functions CVS, CVOS, CVE, CVF, and CVG (see page 40 and following). If this number (WIDTH) is positive, enough blanks will be inserted in front of the resultant string to make the entire results at least WIDTH characters long. The sign, if any, will appear after the blanks. If WIDTH is negative, leading zeroes will be used in place of blanks, The sign, of course, will appear before the zeroes This parameter is initialized by the system to 0

In addition, the DIGITS parameter allows one to specify the number of digits to appear following the decimal point in strings created by CVE, CVF, and CVG. This number is initially 7. See the writeups on these functions for details.

NOTE: All type conversion routines, including those that SETFORMAT applies to, are performed at compile time if their arguments are constants. However, Setformat does not have its effect until execution time. Therefore, CVS, CVOS, CVE, CVF, and CVG of constants will have the no leading zeros and 7 digits (if any) following the decimal point.

GETFORMAT

GETFORMAT (@WIDTH , @DIGITS);

The WIDTH and DIGIT settings specified in the last SETFORMAT call are returned in the appropriate reference parameters.

CVS

"ASCII-STRING" ← CVS (VALUE);

The decimal Integer representation of VALUE is produced as an ASCII String with leading zeroes omitted (unless WIDTH has been set by SETFORMAT,page 40, to some negative value). "-" will be concatenated to the String representing the decimal absolute value of VALUE if VALUE is negative.

CVOS

"ASCII-STRING" ← CVOS (VALUE);

The octal Integer representation of VALUE is produced as an ASCIi String with leading zeroes omitted (unless WIDTH has been set to some negative value by SETFORMAT,page 4Ø). No "-" will be used to indicate negative numbers. For instance, -5 will be represented as "777777777773".

CVE, CVF, CVG

"STRING" ← CVE (VALUE);
"STRING" ← CVF (VALUE);
"STRING" ← CVG (VALUE);

Real number output is facilitated by means of one of three functions CVE,CVG, or CVF, corresponding to the E,G, and F formats of FORTRAN IV. Each of these functions takes as argument a real number and returns a string. The format of the string is controlled by another function SETFORMAT (WIDTH,DIGITS) (see page 4Ø) which is used to change WIDTH from zero and DIGITS from 7, their initial values. WIDTH specifies the minimum string length. If WIDTH is positive leading blanks will be inserted and if negative leading zeros will be inserted.

The following table indicates the strings returned for some typical numbers. _ indicates a space and it is assumed that WIDTH←1Ø and DIGITS-3.

CVF	CVE	CVG
______.000	__.100@-3_	__.100@-3_
______.001	__.100@-2_	__.100@-2_
______.010	__.100@-1_	__.100@-1_
______.100	__.100____	__.100____
_____1.000	__.100@1__	__1.00____
____10.000	__.100@2__	__10.0____
-100.000	__.100@3__	__100.____
-1000.000	__.100@4__	__.100@4__
-10000.000	__.100@5__	__.100@5__
-100000.000	__.100@6__	__.100@6__
-1000000.000	__.100@7__	__.100@7__
-1000000.000	_-.100@7__	_-.100@7__

The first character ahead of the number is either a blank or a minus sign. With WIDTH←-1Ø plus and minus 1 would print as:

```
    CVF          CVE           CVG
 _00001.000   _0.100@1__     _01.00____
 -00001.000   -0.100@1__     -01.00____
```

All numbers are accurate to one unit in the eighth digit. If DIGITS is greater than 8, trailing zeros are included; if less than eight, the number is rounded.

CVSTR

"STRING" ← CVSTR (VALUE);

VALUE is treated as a 5-character left-justified word full of ASCII. the result is a 5-character long String containing these characters. The low order bit of VALUE is ignored.

CVXSTR

"STRING" ← CVXSTR (VALUE);

VALUE is treated as a 6-character left-justified word full of SIXBIT. The result is a 6-character long String containing these characters, converted to ASCII.

CVD

VALUE ← CVD ("ASCII-STRING");

ASCII-STRING should be a String of decimal ASCII characters perhaps preceded by plus and/or minus signs. Characters with ASCII values ≤ SPACE ('40) are ignored preceding the number. Any character not a digit will terminate the conversion (with no error indication). The result is a (signed) integer.

CVO

VALUE ← CVO ("ASCII-STRING");

This function is the same as CVD except that the input characters are deemed to represent Octal values.

CVASC

VALUE ← CVASC ("STRING");

This is the inverse function for CVSTR. Up to five ASCII characters

CVSIX

VALUE ← CVSIX ("STRING");

The inverse for CVXSTR, this function works the same as CVASC except that up to six SIXBIT characters are placed in VALUE. The characters from STRING are converted from ASCII to SIXBIT before depositing them in VALUE.

8.2 - STRING MANIPULATION ROUTINES

EQU

VALUE ← EQU ("STR 1","STR2");

The value of this function is TRUE if STR1 and STR2 are equal in length and have identically the same characters in them (in the same order). The value of EQU is FALSE otherwise.

LENGTH

VALUE ←LENGTH("STRING")

LENGTH is always an integer-valued function. If the argument is a String, its length is the number of characters in the string. The length of an algebraic expression is always 1 (see page 21). LENGTH is usually compiled in line.

LOP

VALUE ← LOP (STRINGVAR)

The LOP operator applied to a String variable removes the first character from the String and returns it in the form given in page 21 above. The String no longer contains this character. LOP applied to a null String has a zero value. LOP is usually compiled in line.

8.3 - LIBERATION-FROM-SAIL ROUTINES

CODE

RESULT ← CODE (INSTR , @ADDR)

This function is equivalent to the FAIL statements:

```
EXTERNAL   .SKIP.   ;DECLARE AS _SKIP_ IN SAIL
SETOM      .SKIP.   ;ASSUME SKIP
MOVE       0,INSTR
ADDI       0,@ADDR
XCT        0
SETZM      .SKIP.   ;DIDN'T SKIP
RETURN     (1)
```

In other words, it executes the instruction formed by adding the address of the ADDR variable (passed by reference) to the number INSTR. Before the operation is carried out, AC1 is loaded from a special cell (initially Ø). AC1 is returned as the result, and also stored back into the special cell after the instruction is executed. The global variable _SKIP_ (SKIP. in DDT or FAIL) is FALSE (Ø) after the call if the executed Instruction did not skip; TRUE (currently -1) if it did. Declare this variable as EXTERNAL INTEGER _SKIP_ if you want to use it.

CALL

RESULT ← CALL (VALUE , "FUNCTION");

This function is equivalent to the FAIL statements:

```
EXTERNAL   .SKIP.
SETOM      .SKIP.
MOVE       1,VALUE
CALL       1, [SIXBIT /FUNCTION/]
SETZM      .SKIP.        ;DID NOT SKIP
RETURN     (REGISTER 1)
```

The SKIP. variable (_SKIP_ in SAIL) is set as described in the previous paragraph (CODE)

USERCON

USERCON(@INDEX , @VALUE , FLAG)

This function allows inspection and alteration of the "User Table". The user table is always loaded with your program and contains many interesting variables. Declare an index you are interested in as an External Integer (e.g., EXTERNAL INTEGER REMCHR). This will, when loaded, give an address which is secretly a small Integer Index Into the User Table. When passed by reference, this index is available to USERCON. The names and meanings of the various User Table indices can be found in the file HEAD, wherever SAIL compiler program text files are sold.

USERCON always returns the current value of the appropriate User Table entry (the Global Upper Segment Table is used if FLAG is negative and your system knows about such things). If FLAG is odd, the contents of VALUE before the call replaces the old value in the selected entry of the selected table.

By now the incredible danger of this feature must be apparent to you. Be sure you understand the ramifications of any changes you make to any User Table value.

USERERR

USERERR(VALUE , CODE , "MSG" , "RESPONSE");
USERERR(VALUE , CODE , "MSG");

USERERR generates an error message. See page 95 for a description of the error message format. MSG is the error message that is printed on the teletype or sent to the log file. If CODE = 2, VALUE is printed in decimal on the same line. Then on the next line the "LAST SAIL CALL" message may be typed which indicates where in the user program the error occurred. If CODE is 1 or 2, a "→" will be typed and execution will be allowed to continue. If it is 0, a "?" is typed, and no continuation will be permitted. The string RESPONSE, if included in the USERERR call, will be scanned before the input buffer is scanned. In fact, if the string RESPONSE satisfies the error handler, the Input buffer will not be scanned at all. Examples:

USERERR(Ø,1,"LINE TOO LONG"); Gives
error message and allows continuation.

USERERR(Ø,1,NULL,"QLA"); Resets mode
of error handler to Quiet, Logging, and
Automatic continuation. Then continues.

8.4 - BYTE MANIPULATION ROUTINES

LDB, DPB, etc.

VALUE ←LDB(BYTE-POINTER);
VALUE ←ILDB(@ BYTE-POINTER);
DPB (BYTE, BYTE-POINTER);
IDPB(BYTE, @ BYTE-POINTER);
IBP (@ BYTE-POINTER);

LDB, ILDB, DPB, IDBP, and IBP are SAIL constructs used to invoke the PDP-10 byte loading instructions. The arguments to these functions are expressions which are interpreted as byte pointers and bytes. In the case of ILDB, IDPB, and IBP, you are required to use an algebraic variable as argument as the

byte-pointer, so that the byte pointer (i.e. that algebraic variable) may be incremented.

POINT

VALUE ← POINT (BYTE SIZE ,
@EFFECTIVE ADDRESS , LAST BIT NUMBER

POINT returns a byte pointer (hence it is of type Integer). The three arguments are enough to specify the three fields of a POP-10

If the LAST BIT NUMBER is -1, POINT creates a byte pointer which, when used with an ILDB, will pick up the first byte from the word at EFFECTIVE ADDRESS, Otherwise, the three arguments to POINT are exactly analogous to the three arguments to POINT in FAIL.

8.5 - OTHER USEFUL ROUTINES

CVFIL

VALUE ← CVFIL ("FILE-SPEC" , @EXTEN , @PPN);

FILE-SPEC has the same form as a file name specification for LOOKUP or ENTER. The SIXBIT for the file name is returned in VALUE. SIXBIT values for the extension and project-programmer numbers are returned in the respective reference parameters. Any unspecified portions of the FILE-SPEC will result in zero values. The global variable _SKIP_ (accessed by declaring it as EXTERNAL INTEGER _SKIP_) will be 0 if no errors occurred, ≠Ø if an invalid file name specification is presented.

FILEINFO

FILEINFO (@INFOARRAY);

FILEINFO fills the 6 word array designated by the array name that is its argument with the following six words from the most recent LOOKUP, ENTER, or RENAME:

Project,,programer name (in SIXBIT)
filename (in SIXBIT)
extension (in SIXBIT)
date last written
date last dumped
protection
size

ARRINFO

VALUE ← ARRINFO (ARRAY, PARAMETER);

ARRINFO(ARRAY,-1) returns the number of dimensions for the array. This number is negative for String arrays.

ARRINFO(ARRAY,Ø) returns the total size of the array in words.

ARRINFO(ARRAY,1) returns the lower bound for the first dimension.

ARRINFO(ARRAY,2) returns the upper bound for the first dimension.

ARRINFO(ARRAY,3) returns the lower bound for the second dimension.

ARRINFO(... etc.

ARRBLT

ARRBLT (@LOC1, @LOC2 , NUM);

NUM words are transferred from consecutive locations starting at LOC2 to consecutive locations starting at LOC1. No bounds checking is performed. This function does not work well for String Arrays (nor set nor list arrays).

ARRTRAN

ARRTRAN (ARRAY 1, ARRAY2);

This function copies information from ARRAY2 to ARRAY 1. The transfer starts at the first data word of each array. The minimum of the sizes of ARRAY 1 and ARRAY2 is the number of words transferred.

IN-CONTEXT

VALUE ← IN-CONTEXT (VARI , CONTXT);

IN-CONTEXT is a boolean which tells one if the specified variable is in the specified context. VARI may be any variable, array element, array name, or Leap variable. If that variable, element or array was REMEMBERed in that context, IN-CONTEXT will return True. IN-CONTEXT will also return true if VARI is an array element and the whole array was Remembered in that context (by using REMEMBER <array_name>). On

the other hand, if VARI is an array name, then IN-CONTEXT will return true only if one has Remembered that array with a REMEMBER <array-name>.

SECTION 9

MACROS AND CONDITIONAL COMPILATION

9.1 - SYNTAX

<define>
::= DEFINE <def_list> ;
::= REDEFINE <def_list> ;
::= EVALDEFINE <def_list> ;

<def_list>
::= <def>
::= <def_list> , <def>

def>
::= <identifier> = <macro-body>
::= <Identifier> (<id-list>) =
<macro-body>
::= <identifier> <string_constant> =
<macro-body>
::= <identifier> (<id-list>)
<string-constant> = <macro_body>

<macro-body>
::= <delimited_string>
::= <constant-expression>
::= <macro-body> & <macro-body>

<macro_call>
::= <macro_identifier>
::= <macro_identifier> (
<macro_param_list>)
::= <macro-identifier> <string-constant>
(<macro_param_list>)

<macro_identifier>
::= <identifier>

<macro_param_list>
::= <macro_param>
::= <macro_param_list> , <macro_param>

<cond_comp_statement>
::= <conditional_c.c.s.>
::= <while_c.c.s.>
::= <for_c.c.s.>
::= <for_list_c.c.s.>
::= <case_c.c.s.>

<conditional_c.c.s.>
::= IFC <constant-expression> THENC
<anything> ENDC
::= IFC <constant-expression> THENC
<anything> ELSEC <anything> ENDC
::= IFCR <constant-expression> THENC
<anything> ENDC
::= IFCR <constant-expression> THENC
<anything> ELSEC <anything> ENDC

<while_c.c.s.>
::= WHILEC <delimited_expr> DOC
<delimited-anything> ENDC

<for_c.c.s.>
::= FORC <constant-expression> ←
<constant-expression> STEPC
<constant-expression> UNTILC
<constant-expression> DOC
<delimited-anything> ENDC

<for_list_c.c.s.>
::= FORLC <identifier> ←
(<macro_param_list>) DOC
<delimited_anything> ENDC

<case_c.c.s.>
::= CASEC <constant-expression> OFC
<delimited-anything-list> ENDC

<delimited-anything-list>
::= <delimited-anything>
::= <delimited_anything_list> ,
<delimited_anything>

<assignc>
::= ASSIGNC <identifier> = <macro-body> ;

<delimited_string>, <macro_param>, <delimited_expr>, <anything> and <delimited_anything> are explained in the following text.

9.2 - DELIMITERS

There are two types of delimiters used by the Sail macro scanner: macro body delimiters and macro parameter delimiters. Their usage will be precisely defined in the sections on Macro Bodies and Parameters to Macros, Here we will discuss their declaration and scope, which is very important when using source files with different delimiters (see page 10 to find out about source files).

Sail initializes both left and right delimiters of both body and parameter delimiters to the double quote ("). One may change delimiters by saying

```
REQUIRE "⊂⊃<>" DELIMITERS.
```

In this example, the left and right body delimiters become "⊂" and "⊃", while the left and right parameter delimiters become "<" and ">". Require Delimiters may appear wherever a statement or declaration is legal, One should Require Delimiters whenever all but the most simple macros are going to be used. The first Require Delimiters will initialize the macro facility; if this is not done, some of the following conveniences will not exist and only very simple macros like defining CRLF = "(' 12 & ' 15)" may be done.

Delimiters do not follow block structure. They persist until changed. Furthermore, each time new delimiters are Required, they are stacked on a special "delimiters stack". The old delimiters may be revived by saying

```
REQUIRE UNSTACK_DELIMITERS
```

Thus, each source file with macros should begin with a Require delimiters, and end with an Unstack-delimiters. It is impossible to Unstack off the bottom of the stack. The bottom element of the stack is the double quote delimiters that Sail initialized the program to. If you Unstack from these, the Unstack will become a no-op, and the double quote delimiters remain the delimiters of your program.

One may circumvent the delimiter stacking feature by saying

```
REQUIRE "⊂⊃<>" REPLACE_DELIMITERS
```

instead of REQUIRE "⊂⊃<>" DELIMITERS. This doesn't deactivate the stacking feature, it merely changes the active delimiters without stacking them.

To revert to the primitive, initial delimiter mode where double quotes are the active delimiters, one may say

```
REQUIRE NULL DELIMITERS
```

Null delimiters are stacked in the delimiter stack in the ordinary REQUIRE "⊂⊃<>" DELIMITERS way. In null delimiters mode, the double quote character may be included in the macro body or macro parameter by "using two double quotes:

```
DEFINE SOR = "OUTSTR(""SORRY"");";
```

The Null Delimiters mode is essentially the macro facility of older versions of Sail where " was the only delimiter. Programs written in older Sail versions will run in Null Delimiters mode. Null delimiters mode has all the rules and quirks of the old Sail macro system (the old Sail macro facility is described in [Swinehart 8 Sproull], Section 13). Compatibility with the old Sail is the only reason for Null Delimiters.

9.3 - MACROS

We will delay the discussion of macros with parameters until the next section. A macro without parameters is declared by saying:

```
DEFINE <macro-name> = <macro_body>;
```

where <macro-name> is some legal identifier name (see page 89 for a definition of a legal identifier name). <macro_body>s can be simply a sequence of Ascii characters delimited by macro body delimiters, or they can be quite complex. Once the macro has been defined, the macro body is substituted for every subsequent appearance of the macro name. Macros may be called in this way at any point in a Sail program, except inside a Comment or a string constant.

Macro declarations may also appear virtually anywhere in a Sail program. When the word DEFINE is scanned by Sail, the scanner traps to a special production. The Define is parsed, and the scanner returns to its regular mode as if there had been no define there at all. Thus things like

```
I ← J + 5 + DEFINE CON = ⊂'777⊃;K↑2;. . .
```

are perfectly acceptable. However, don't put a Define in a string constant or a Comment.

SCOPE

Macros obey block structure. Each DEFINE serves both as a declaration and an assignment of a macro

body to the newly declared symbol. Two DEFINEs of the same symbol in the at the same lexical level will be flagged as an error. However, it is possible to change the macro body assigned to a macro name without redeclaring the name by using saying REDEFINE instead of DEFINE. For example,

```
BEGIN
...
 BEGIN
 ...
 DEFINE SQUAK = ⊂OUTSTR("OUTER BLOCK");⊃;
 ...
  BEGIN
  ...
  REDEFINE SQUAK = ⊂OUTSTR("INNER BLOCK");⊃;
  ...
  END;
 ...
 SQUAK COMMENT Here the program types
         "INNER BLOCK";
 END;  COMMENT Here SQUAK is undefined.
       If SQUAK were included here, you'd
       get the error message
          "UNDEFINED IDENTIFIER:SQUAK";
END
```

REDEFINE of 3 name that has not been declared in a DEFINE will act as a DEFINE That is, it will also declared the macro name as well as assigning a body to it.

MACRO BODIES
A Macro Body may be

1. A sequence of Ascii characters preceded by a left macro body delimiter and followed by a right macro body delimiter.

2. An integer expression that may be evaluated at compile time.

3. A string expression that may be evaluated at compile time.

4. Concatenations of the above.

WARNING: Source file switching inside macros will not work.

DELIMITED STRINGS
Any sequence of Ascii characters, including " may be used as a macro body if they are properly delimited. The macro body scanner keeps a count of the number of left and right delimiters seen and will terminate its scan only when it has seen the same number of each. This lets the macro body delimiters "nest" so that one may include DEFINEs inside a macro body. For example,

```
DEFINE DEF =
  ⊂DEFINE SYM = ⊂SYMBOL⊃; SYM⊃ ;
```

One may temporarily override the active delimiters by including a two character string before the "=" of the Define statement. For example:

```
DEFINE LES "&%" = & 0≤X<BIGGEST ∧ Y>X %;
```

The first character of the two character string becomes the left delimiter, and the second becomes the right delimiter.

INTEGER COMPILE TIME EXPRESSIONS
Sail tries to do as much arithmetic as it can at compile time. In particular, if you have an arithmetic expression of constants, such as

```
91.504 t (3.1415*8↑(9-7))
      % "Sail can convert strings"
```

then the whole expression will be evaluated at compile time and the resultant constant, in this case 93.9263610, will be used in your code instead of the constant expression. Runt ime functions of constants will be done at compile time too, if possible. EQU and the conversion routines (CVS, CVO, etc.) will work.

When an integer compile time expression is scanned as part of a macro body, it is immediately evaluated. The integer constant which results is converted to a character string, and that character string used for the place in the macro body of the integer expression. Thus,

```
DEFINE TTYUUO = '51 LSH 30 ;
```

will cause '51 LSH 30 to be evaluated, and the resulting constant, -2469606 152, will be converted to the character string -2469606152, and that character string assigned to the macro name TTYUUO.

STRING COMPILE TIME EXPRESSIONS
If a compile time expression has the type string (constant), the macro scanner will evaluate the expression immediately. However, the string constant that results will not be converted to the character string that represents that constant, but to the character string with the same characters that the string constant had. Thus, the way to use a macro for string constants is to delimit the string constant like this:

```
DEFINE STRINCON = ⊂"Very long
   complex string that is hard
      to type more than once"⊃ ;
```

However, the automatic conversion of string constants to character strings is helpful and indeed essential for automatic generat ion of ident if iers:

```
DEFINE N = 1;
        COMMENT we will use this like a variable;

 DEFINE GENSYM = c
    DEFINE SYM = cTEMP_⊃&CVS(N);
        COMMENT SYM is defined to be the character
        string TEMP_# where # is an number;

    DEFINE N = N+1;
        COMMENT This increments N;

    SYM ⊃;
        COMMENT At the call of SYM, the character
        string is read like program text. E. g. . . ;

 INTEGER GENSYM, GENSYM, GENSYM, GENSYM;
 REAL GENSYM, GENSYM;
        COMMENT We have generated 6 identifiers with
        unique names, and declared 4 as integers,
        2 as reals;
```

To convert a macro body to a string constant, one may call the runtime CVMS:

```
<string constant> ←CVMS( <macro name>)
```

A string that has the exact same characters as the macro body will be returned. For example:

```
DEFINE A = cB & C⊃;
DEFINE ABC = CVMS(A) & c &D⊃;
 COMMENT ABC now stands for the text B & C & D;
```

HYBRID MACRO BODIES

When two delimited strings are concatenated, the result is a longer delimited string. "&" in compile time expressron behaves the same way it behaves in any expression. When a compile time expression is concatenated to a delimited character string in a macro body, the result is exactly the result one would get if the delimited character string were a string constant, except that the result is a delimited character string. For exampie:

```
DEFINE N = 1;
DEFINE M = 2;
DEFINE SYM -CVS(N*M t N↑2) & c-SQRT(N*M+1)⊃;
DEFINE SYM1 = c3-SQRT(N*M+1)⊃;
```

Here SYM is exactly the same as SYM1.

9.4 - MACROS WITH PARAMETERS

One defines a macro with parameters by specifing the formal parameters in a list following the macro name:

```
DEFINE MAC (A,B)=cIF A THEN B ELSE ERR←1;⊃;
```

One calls a macro with parameters by including a list of delimited character strings that will be substituted for each occurance of the corresponding formal in the macro body. For example,

```
COMMENT we assume that "<" and ">" are the
        parameter delimiters at this point;
MAC ( BYTES LAND (BITMASK+'2000)> , <
        BEGIN
          WWDAT ←FETCH(BYTES, ENVIRON):
          COLOR[WWDAT] ←' 2000;
        END >)
```

expands to

```
IF BYTES LAND (BITMASK+' 2000) THEN
        BEGIN
          WWDAT ←FETCH(BYTES,ENVIRON);
          COLOR[WWDAT]←' 2000;
        END
        ELSE ERR←1;
```

Parameter delimiters nest. Furthermore, if no delimiters are used about a parameter, nesting counts are kept of "()", "[I", and "{}" character pairs. The parameter scan will not terminate until the nesting counts of each of the three pairs is zero. One may temporarily override the active parameter deiimiters by including a two character string ahead of the parameter list in the macro call:

```
MAC "‹∃"(‹BYTES>' 20003, ‹MATCH(BYTES)∃)
```

Formal parameters may not appear in compile time expressions that are used to specify macro bodies. This is quite natural: compile time expressions must be evaluated as they are scanned, but the value of a formal parameter isn't known until later. However, if the macro body is a hybrid of expressions and delimited character strings, then formal parameters may appear in the delimited string parts.

When doing a CVMS on a macro with parameters, use only the macro name in the call; the parameters are unnecessary. The string returned will have the two character strings "∝1", "∝2", etc. (here ∝ stands for the Ascii character '177) where the formal parameters were in the macro body. A "∝1" will appear wherever the first formal parameter of the formal parameter list appear in the macro body, a "∝2" will appear wherever the second parameter appeared, etc. The unfortunate appearance of the Ascri character '177 in CVMS generated strings is a product of the representation of macro bodies as strings (ending in '177, '0 which CVMS removes) having '177,(n+'61) for each appearances of the nth formal parameter in the body.

9.5 - CONDITIONAL COMPILATION

The compile time equivalents of the Sail IF, WHILE, FOR and CASE statements are

```
IFC <CT expr> THENC <anything> ENDC

IFC <CT expr> THENC <anything> ELSEC
      <anything> ENDC

WHILEC ⊂<CT expr>⊃ DOC ⊂<anything>⊃ ENDC

FORC <CT variable> ← <CT expr> STEPC <CT expr>
      UNTILC <CT expr> DOC ⊂<anything>⊃ ENDC

FORLC <CT variable> ← (<macro param>, . . . ,
      <macro param>) DOC ⊂<anything>⊃ ENDC

CASEC <CT expr> OFC ⊂<anything>⊃, ⊂<anything>⊃,
      . . . , ⊂<anything>⊃ ENDC
```

where <CT expr> is any compile time expression. <CT expr> could itself include IFCs, FORCs or whatever. <CT variable> is a macro name such as N from a define such as DEFINE N = MUMBLE: <macro param> is anything that is delimited like a macro parameter. <anything> can be anything one could want in his program at that point, including Defines and other conditional compilation statements. The usual care must be taken with nested IFCs so that the ELSECs match the desired THENCs. The "⊂" and "⊃" characters above are to stand for the current MACRO BODY DELIMITER pair.

The semantics are exactly those of the corresponding runtime statements, with one exception When the list to a FORLC is null (i.e. it looks like "()"), then the <anything> is inserted in the compilation once, with the <CT variable> assigned to the null macro body.

Situations frequently occur where the false part of an IFC must have the macros in it expanded in order to delimit the false part correctly. For example,

```
DEFINE DEBUG-SELECT =
  ⊂IFC DEBNUM = 2 THENC ⊃;
DEFINE DEBUG-END =
  ⊂ELSEC OUTSTRYDEBUG POINT") ENDC⊃;

Debug-select
    OUTSTR("DEBUG POINT #" & CVS(DBN));
Debug-end
```

If DEBNUM is not 2, then the program must expand the macro Debug-end in order to pick up the ELSEC that terminates the false part of the conditional. The expansion is only to pick up such tokens -- the text of the false part is not sent to the scanner as the true part is. In order to avoid such expansion, one may use IFCR (the R stands for "recursive") instead of IFC.

As an added feature, when delimiters are required about an <anything> in the above (such constructs are named <delimited-anything> in the BNF), one may substitute a concatenation of constant expressions and delimited strings. This is just like a macro body, except the concatenation MUST contain at least one delimited string, thereby forcing the result of the concatenation to be a delimited string, rather than a naked expression.

As a further added feature,

```
IFC <CT expr> THENC ⊂<anything>⊃ ELSEC
    ⊂<anything>⊃ ENDC
```

may be substituted in FORCs, FORLCs, and WHILECs for the <anything> following DOC.

NOTE: In a WHILEC, the expression must be delimited with the appropriate macro body delimiters (hence the construct <delimited_expr> in the BNF).

9.6 - TYPE DETERMINATION AT COMPILE TIME

To ascertain the type of an identifier at compile time, one may use the integer function DECLARATION <identifier>). This returns an integer with bits turned on to represent the type of identifier. Exactly what the bits represent is a dark secret and changes periodically anyway. The best way to decode the integer returned by Declaration is to compare it to the integer returned by CHECK_TYPE(<a string of Sail declarators>). A Sail declarator is any of the reserved words used an a declaration. Furthermore, the declarators must be listed in a legal order, namely, an order that is legal in declarations (i.e. ARRAY INTEGER won't work). One may include as arguments to CHECK-TYPE the following special tokens:

TOKEN	EFFECT
BUILT-IN	The bit that is on when an identifier is predeclared by Sail, such as CVS, NULL, etc. is returned.
LPARRAY	The bit that is on when an identifier is an item or itemvar with a declared array datum is returned (the discussion of Leap starts on page 51).
RESERVED	The bit that indicates the identifier word is returned.
DEFINE	The bit that indicates the identifier is a

macro name is returned (note: a macro name as the argument DECLARATION will not be expanded).

Examples:

DECLARATION(FOO) = CHECK_TYPE(INTEGER)
This is an exact compare. Only if Foo is an integer variable will equality hold.

DECLARATION(A) LAND CHECK_TYPE(ARRAY)
This is not an exact compare. If A is any kind of an array, the LAND will be non-zero.

DECLARATION(CVS) = CHECK_TYPE(BUILT_IN STRING PROCEDURE)
The equality holds.

DECLARATION(BEG) LAND CHECK_TYPE(RESERVED)
This is non-zero only if one has said LET BEG = BEGIN. DEFINE BEG = BEGIN will only turn the Define bit of BEG on.

NOTE: if the <identifier> of DECLARATION has not yet been declared or was declared in an inner block, then 0 is returned -- it is undeclared so it has no type.

9.7 - MISCELANEOUS FEATURES

COMPILE TIME I/O
Compile time input is handled by the REQUIRE "<file-name>" SOURCE-FILE construct. <file-name> can be any legal file, including TTY: and MTAO: and of course disk files. The file will be read until the its end of file delimiter is scanned (<cntr>Z for TTYs or <meta><cntr><lf> at Stanford), and its text will replace the REQUIRE statement in the main file.

Compile time output is limited to typing a message on the users teletype. To do this say REQUIRE <string-constant> MESSAGE, and the <string-constant> will appear on your teletype when the compilation hits that point in your file.

EVALDEFINE
The reserved word EVALDEFINE may be used in place of the word DEFINE if one would like the identifier that follows to be expanded. When one follows a DEFINE with a macro name, the macro is not expanded, but rather the macro name is declared at the current lexical level and assigned the specified macro body. EVALDEFINE gets you around that. Helps with automatic generation of macro names.

ASSIGNC
The following compile time contruct makes recursive macros easier.

ASSIGNC <name1> = <macro-body>;

<name1> must be a formal to a macro, and <macro_body> may be any macro body. Thereafter, whenever <name1> is instantiated, the body corresponding to <macro-body> is used in the expansion rather than the text passed to the formal at the macro call.

RESTRICTION. ASSIGNC may only appear in the body of the macro that <name1> is a formal of. If it appears anywhere else, the <name1> will be expanded like any good formal, and that text used in the ASSIGNC as <name1>. Unless you're being very clever, this is probably not what you want.

NOMAC
Preceding anything by the token NOMAC will inhibit the expansion of that thing should that thing turn out to be a macro.

SECTION 10

LEAP DATA TYPES

10.1 - INTRODUCTION

In addition to the standard algal-like statements and expressions, SAIL contains an associative data store and auxiliary facilities called LEAP. SAIL's version of LEAP is based on the associative components of the LEAP language implemented by J. Feldman and P. Rovner as described in [Feldman].

An associative store allows the retrieval of data based on the partial specification of that data. LEAP stores associative data in the form of ASSOCIATIONS, which are ordered three-tuples of ITEMS. Associations are frequently called TRIPLES. Associations are placed in the associative store by MAKE statements and removed from the store by ERASE statements. The associative searches allow us to specify items and their position in the triple and then have the LEAP interpreter search for triples in the associative store which have the same items in the same positions. The interpreter will extract the items from such triples, which correspond to the positions left unspecified in the original search request. For example say we had triples representing the binary relation Father-of, and we had "made" associations of the form

Father-of ⊗ John ≡ Tom
Father-of ⊗ Tom ≡ Harry,
Father-of ⊗ Jerry ≡ Tom,

where Father-of, John, Tom, Harry, and Jerry are names of items. We could then perform searches to find the sons of Tom by specifying to the leap search routines that we wanted to find triples whose first component was Father-of and whose third component was Tom. Associative searches Inherently produce multiple values (i.e. both Jerry and John are sons of Tom). To deal with multiple values, Leap has SETs and LISTs of items.

Items are constants. They may be created by declaration or by the function NEW. Items may have a single algebraic variable, set, list or array associated with them which is accessible by use of the DATUM construct Declared items have names which may be used to identify them in expressions, etc. The simple variable whose value is an item is called an ITEMVAR.

10.2 - SYNTAX

The following syntax is meant to REPLACE not supplement the syntax of algebraic declarations, except where noted.

```
<declaration>
    ::= <type-declaration>
    ::= <array-declaration>
    ::= <procedure-declaration>
    ::= <label-declaration>
    ::= <synonym-declaration>
    ::= <preload_specification>
    ::= <require-specification>
    ::= <context-declaration>
    ::= <type-qualifier> <declaration>


<simple-type>
    ::= REAL
    ::= INTEGER
    ::= STRING
    ::= BOOLEAN
    ::= SET
    ::= LIST


<itemvar_type>
    ::= ITEMVAR
    ::= <simple_type> ITEMVAR
    ::= <array_type> ARRAY ITEMVAR
    ::= CHECKED <itemvar_type>


<item_type>
    ::= ITEM
    ::= <simple-type> ITEM


<array_type>
    ::= <simple-type>
    ::= <itemvar_type>
    ::= <item-type>


<type-declaration>
    ::= <simple-type> <identifier-list>
    ::= <itemvar_type> <identifier-list>
    ::= <item-type> <identifier-list>
    ::= <array-type> ARRAY <array-list>
    ::= <array-type> ARRAY ITEM <array-list>
    ::= <type-qualifier> <type-declaration>


<array_list> -- as on page 3
```

<procedure_declaration>
::= PROCEDURE <Identifier> <procedure-head> <procedure-body>
::= <procedure-type> PROCEDURE <identifier> <procedure_head> <procedure-body>
::= <type-qualifier> <procedure-declaration>

<procedure-type>
::= <simple-type>
::= <itemvar_type>
::= MATCHING <procedure-type>

<procedure-head> and <procedure-body> -- as on page 4 except:

<simple-formal-type> ::= <simple_type>
<itemvar_type>
::= ? <itemvar_type>
::= <simple-type> ARRAY
::= <itemvar_type> ARRAY
::= <simple-type> PROCEDURE
::= <itemvar_type> PROCEDURE

<preload_specification>, <synonym-declaration>, <label-declaration>,

and <require_specification> as on page 3

<context-declaration> as on page 29

10 3 - SEMANTICS

ITEM GENESIS

Although items are constants, they must be created before they can be used. Items may be created in three ways:

1) A Declared item may created by declaration of an identifier to be of type ITEM.

2) An item may be created with the NEW construct (see page 64).

3) A bracketed triple item is created by the MAKEing of a bracketed triple (see MAKE, page 57).

Items of type 1 and 2 are the same except those of type 1 may be referred to by the identifier that is associated with them. For example one may say ... ITEM DAD; X←DAD; . . . . NOTE: DAD is only the name of an item, not a variable! Saying DAD+-X is just as illegal as saying 15-X.

Items of type 3 are different from those of type 1 and 2. Discussion of them will be left until the creation of associations with the MAKE statement is discussed (page 57).

SCOPE OF ITEMS

Items do not obey the traditional Algol scope rules. All declared items are allocated in the outer block All other items are allocated dynamically. All items exist until a DELETE(<item expression>) is done on them (see page 56 for the details of DELETE), or until the outer block is exited at the end of the program. HOWEVER, the identifiers of declared items (type 1 above) DO obey scope rules. After exiting the block in which item X was declared, it will be Impossible to refer to X by its declared name. However, X may have been stored in an itemvar, associations, etc. and thus still be retrieved and used.

Warning: items in recursive procedures behave differently from variables in recursive procedures. At each recursive call of a procedure, the local variables are reinstantiated (unless they were declared OWN). Items are constants. There is never more than one instantiation of an item around at a time.

DATUMS OF ITEMS

An item of type 1 or 2 may have an associated variable, called its DATUM. The Datum of an item is like any variable; it may be declared to have any type that a variable may have, except the type Itemvar. Because an item may have only one datum from its creation until its death, we frequently will say the "type of an Item" referring to the type of the datum. RESTRICTIONS: It is currently impossible to make either items or their datums either Internal or External. However, the effect of External items can be duplicated by manipulating the order in which items are declared (see page 54). OWN is not applicable as items are constants, not variables, Items of type ARRAY must be declared with constant bounds since they are allocated upon entering the outer block.

Example declarations of items with datums:

```
INTEGER ITEM FATHER-OF;
STRING ITEM FOO;
INTEGER ARRAY ITEM NAMES [1:4,1:8]; COMMENT note
      the specification of the array's dimensions;
SHORT REAL ITEM POINT;
EXTERNAL ITEM BLAT;         COMMENT illegal;
ITEMVAR ITEM BLAT;          COMMENT illegal;
STRING ITEMVAR ITEM BLAT; COMMENT illegal;
REAL PROCEDURE ITEM BLAT; COMMENT illegal;
PROCEDURE ITEM BLAT;        COMMENT illegal,
                            use ASSIGN;
```

The syntax for variable includes the Datum construct. That is, if AGE is a declared an Integer Item, then DATUM(AGE) behaves exactly like an Integer variable. If ARR is declared as

```
STRING ARRAY ITEM ARR [2:4,1:9+2]
```

then DATUM(ARR) is a string array with two dimensions of the declared size. A new array may not be assigned to the Datum of ARR, though of course the individual elements of the array may be changed. Datums obey the -same type checking and type conversion rules that the algebraic variables of Sail do. For example, when a string is assigned to an integer datum, the integer stored in the integer datum is the ASCII of the first character of the string.

ITEMVARS

An Itemvar is a variable whose value is an Item. Just as the statements "X←3; Y-X" and "Y←3" are equivalent with respect to Y, the statements "X-DAD; Y-X" and "Y←DAD" are equivalent with respect to Y, if X and Y are itemvars, DAD an item. The distinction between Itemvars and items is identical to the distinction between integer variables and integers. An integer variable may only contain an integer and a variable declared ITEMVAR may only contain an item. This may be confusing since historically, integer variables have always been called INTEGER rather than INTEGERVAR.

Properly speaking, one should have INTEGERVAR ARRAYs Instead of INTEGER ARRAYs. Originally, Sail only allowed ITEMVAR ARRAYs. However, so many people found this confusing that now one may say ITEM ARRAY, and it will be interpreted to mean ITEMVAR ARRAY. Similarly, an Item procedure is exactly the same as an Itemvar procedure.

An ttemvar may contain items of any type. However, when one says DATUM(ITMVR) where ITMVR is an itemvar, the compiler must know the type of the datum of the item (i.e. the type of the item) contained in the Itemvar so that the the correct conversions, etc. may be done. Thus, one may declare itemvars to have the same types that are legal for items. If one has declared STRING ITEMVAR ITMVR, then the compiler assumes that you have stored an string item in ITMVR, and and will treat DATUM(ITMVR) as a string variable.

An Itemvar may be declared CHECKED if the user desires the type of itemvar checked against the type of the datum of the item expressions assigned to it. That is, only a string item could be stored in a Checked String Itemvar. If the itemvar is not declared Checked, it may have an item of any type assigned to it and their types need not match at all. This can be very dangerous. For example, an integer array item might be assigned to a string itemvar. When the datum of this itemvar is later assigned to an integer variable, say, Sail will try to treat the array header as a string pointer and get very confused. The runtime routine TYPEIT, page 83, returns a code for the type of its argument, and can be useful for avoiding type matching errors with un-checked itemvars.

EXTERNAL, OWN and INTERNAL Itemvars are legal. SAFE applies to either the array of an array itemvar, the array of an itemvar array, or both arrays of an array itemvar array.

Itemvars obey traditional Algol block structure. Upon exiting the block of their declaration, their names are unavailable and their storage is reallocated. However, the item stored in an itemvar is not affected -- it continues to exist until DELETEd or until the end of the program.

Itemvars are initialized to the special item ANY at the beginning of one's program.

SETS AND LISTS

Sets and Lists are collections of items. There are two distinctions between Sets and Lists: a list may contain multiple occurrences of any item while a set contains at most a single instance of an item. Second, the order in which items appear within a list is completely within the control of the user program, while with a set, the order is fixed by the internal representation of the items. Lists and Sets do not care what type if any the datums of their members are.

List and Set Arrays, Itemvars, Items, and Procedures are all legal, as well as External, Own and Internal Sets and Lists. Like itemvars, the scope of Set and List variables is the block they were declared in. Exiting that block does not destroy the items stored in the departed sets or lists.

ASSOCIATIONS

Perhaps the most important form of storage of items is the Association, or TRIPLE. Triples of items may be written into or retrieved from a special store, the

associative store. The method of storage of these triples is designed to facilitate fast and flexible retrieval. Sail uses approximately two words of storage for each triple in the associative store. There is at most one copy of a triple in the store at any time. Once a triple has been stored in the associative store, its component items can not be changed, although an approximation to this can be obtained by erasing the association then making a new association with the altered components. You will note there is no syntax for declaring a triple. Triples can only be created with the MAKE statement. In the examples which follow, a triple is represented by :

$$A \otimes O \equiv V$$

where A, 0, and V represent the items stored in the association. The associative store is accessed by the FOREACH statement, derived sets, and binding triples (see Searching the Associative Store, page 57).

PROCEDURES
Itemvar, Item, List, and Set procedures all exist. Itemvar procedures may be CHECKED if one desires the item RETURNed to have the same type as the type of the Itemvar procedure. Otherwise, the compiler only checks to see that the value returned to an itemvar procedure is an item.

Every type except Item may be used in formal parameter declarations; items are constants yet parameters always have something assigned to them in the procedure call. Since you can't assign something to a constant, you can't have item parameters.

WARNING: when using Checked Reference Itemvar formals, no type checking is performed as the actual is assigned to the formal at the procedure call. However, type checking will only be done during the procedure, and when the formal is assigned to the actual upon the (normal) exit of the procedure.

IMPLEMENTATION
Each Item is represented by a unique integer in the complier. The numbers are assigned in the order the items are declared, e.g. the first declared item get 1, the second gets 2, etc. (actually, Sail has already declared 8 items that it needs, so user item numbers start with 9). Lexical nesting is not observed; it is only the sequence in which the declarations are scanned that determines their numbers. The NEW function does not affect this assignment of numbers. Items created by the New function are assigned the next available number at the time of the execution of the New.

Those who use separately compiled procedures (see page 1Ø) may wish to have declared items common to both programs. However, Internal and External items do not exist. The same effect may be-achieved by carefully declaring the desired items in the sarne order in both programs so that their numbers match. The message "Warning -- two programs with items in them." will be issued at the begining of execution, and may be ignored if you are certain the items are declared in the same relative positions. No checking of names, types, arrays bounds, etc. is done, so be very careful.

Items occupy no space (neither does the constant integer 15). The numbers ascribed to items are stored in Itemvars and Associations. Itemvars are simply a word of storage An association is two words of storage, one with three 12 bit bytes, each containing the number of one of the items of the association, and a second word containing two pointers reiating the association to the associative search structure. Since the number of an item must fit in 12 bits, the number of items is limited to about 4090.

The number of an item may be retrieved from the item as a integer with the predeclared function CVN (<item-expression>). The item represented by a certain integer may be retrieved by the predeclared function CVI (<algebraic-expression>). CVN and CVI should only be used by those who know what they're doing and have kept themselves up to date on changes in Leap.

SECTION 11

LEAP STATEMENTS

11.1 - SYNTAX

<leap-statement>
 ::= <leap-assignment-statement>
 ::= <leap-swap-statement>
 ::= <set-statement>
 ::= <list-statement>
 ::= <associative-statement>
 ::= <foreach_statement>
 ::= <suc_fail_statement>

<leap-assignment-statement>
 ::= <itemvar_variable> ← <item-expression>
 ::= <set-variable> + <set-expression>
 ::= <list-variable> + <list-expression>

<leap-swap-statement>
 ::= <itemvar_variable> ↔ <itemvar_variable>
 ::= <set-variable> ↔ <set-variable>
 ::= <list-variable> ↔ <list-variable>

<set-statement>
 ::= PUT <item-expression> IN <set-variable>
 ::= REMOVE <item-expression> FROM
  <set-variable>

<list-statement>
 ::= PUT <item-expression> IN <list-variable>
  <location_specification>
 ::= REMOVE <item-expression> FROM
  <list-variable>
 ::= REMOVE ALL <item_expression> FROM
  <list-variable>

<location_specification>
 ::= BEFORE <element-location>
 ::= AFTER <element-location>

<element_location>
 ::= <item-expression>
 ::= <algebraic_expression>

<associative_statement>
 ::= DELETE (<item-expression>)
 ::= MAKE <triple>
 ::= ERASE <triple>

<triple>
 ::= <item-expression> ⊗ <item-expression> ≡
  <item-expression>

<foreach_statement>
 ::= FOREACH <binding-list> SUCH THAT
  <element-list> DO <statement>
 = NEEDNEXT <foreach_statement>

<binding-list>
 ::= <itemvar_variable>
 ::= <binding-list> , <itemvar_variable>

<element_list>
 ::= <element>
 ::= <element-list> AND <element>

<element>
 ::= <item-expression> IN
  <list-expression>
 ::= (<boolean-expression>)
 ::= <retrieval-triple>
 ::= <matching_procedure_call>

<retrieval-triple>
 ::= <ret-trip-element> ⊗ <ret-trip-element>
  ≡ <ret-trip-element>

<ret-trip-element>
 ::= <item-expression>
 ::= <derived-set>

<matching-procedure-call>
 ::= <procedure-call>

<suc_fail_statement>
 := SUCCEED
 ::= FAIL

11.2 - RESTRICTIONS

SUCCEED and FAIL statements must be lexically nested inside a matching procedure to be legal.

11.3 - SEMANTICS

ASSIGNMENT STATEMENTS
Assignment statements in Leap are similar to those in Algol. Itemvars, Set variables, and List variables may be assigned item, set and list expressions, respectively. Only one automatic coercion is done: a set expression may be assigned to a list variable. NOTE: lists may not be assigned to set variables (use CVSET).

The type of an itemvar is checked against the type of the item expression assigned to it if and only if the itemvar is declared Checked. If a typed item is assigned to an un-Checked itemvar of different or no type, the datum is not affected. Assign an integer item to a string itemvar and the string itemvar will now contain an item with an integer datum. Sail will not know that you have in effect switched the type of the datum and will get very confused if you later try to use the datum of the itemvar; it will treat the integer as a pointer to a two word string descriptor in this case.

DATUM (X) is legal only when X is a typed item expression, namely an item expression that the compiler can discover the type of (not COP(<set>) for example). See page 88 for the BNF of typed item expressions. DATUM (X) is syntactically a variable. It has the type of the typed item expression, X. If X has an array type, then DATUM(X) should be followed by [<subscript_list>]. Appropriate coercions will be done (i.e. string to integer, integer to real, etc.) just as with regular variables in expressions. NOTE: the user is responsible for seeing that the datum of an item expression really is the type that Datum thinks it is (i.e. Datum of a Real Itemvar that has had a string item stored in it will give you garbage).

PROPS (X), where X is an item expression, is legal regardless of the type of X. X may even evaluate to a bracketed triple item, procedure item, or event item. PROPS (X) is syntactically an integer variable. It is limited to integers n where $0 \leq n \leq 4095$. If negative (i.e. two's complement) Integers or integers larger than 4095 are assigned to a PROPS, only the right 12 bits are stored. The rest of the Integer is lost.

PUT
Sets and lists are initially empty. One may put items in them with the PUT statement. "PUT <item expression> IN <set variable>" does exactly what it says.

"PUT <item expression> IN <list variable> BEFORE <algebraic expression>" evaluates the item expression, evaluates the algebraic expression and coerces it into an integer, say n, then puts the item into the list at the nth positron, bumping the old nth item to the n+1th position, and so on down the list. This increases the length of the list by one. "PUT item IN list AFTER n" places the item in the ntlth position and bumps the old nt lth item down to the n+2th position, and so on. If n < 0 or n > (1 t length-of-list), then an error message is given. The special token "∞" may be used in the expression for n to stand for the length of the list.

"PUT <item expression 1> IN <list variable> BEFORE <item expression 2>" cause a search to be made of the list for the item of <item expression 2>. If it is found, the item of <item expression 1> is placed in the list immediately ahead of the item found by the search. "PUT item IN list AFTER item" proceeds the same way, but puts the first item in the list immediately following the second item. If the second item is not an element of the list, a BEFORE will put the first item at the begining of the list, while an AFTER will put it at the end of the list.

REMOVE
To remove an item from a set or list, one may use REMOVE. "REMOVE item FROM set" does just what it says. If the item to be removed from the set does not occur in the set, this statement is a no-op.

"REMOVE n FROM list" removes the nth item from the list. The old ntlth item becomes the nth, and so forth. An error is indicated if n ≤ 0 or n > length-of-list. As before, ∞ should stand for the length of the list. However,

"REMOVE item FROM list" removes the first occurence of the item from the list, If the item is net found, this statement is a no-op.

"REMOVE ALL item FROM list" removes all occurences of the item from the list.

DELETE
Items are represented by unique integer numbers in Sail. Due to the overwhelming desire to store an association in one word of storage, these unique numbers are limited to 12 bits. Thus the total number of items is limited to 4090. The DELETE statement allows one to free numbers for reuse. It is also the only way to get rid of an item short of exiting the program. WARNING: The Delete statement in no way alters the instances of the Deleted items which are present in sets, lists, associations, or itemvars. The user should be sure that there are no instances of the

Deleted item occurring in itemvars, sets, lists or associations. Even saying DELETE(ITMVR) where ITMVR is an itemvar with an item to be deleted in it will not remove the item from ITMVR; one must be careful to change the contents of ITMVR before using it again

MAKE
The MAKE statement is the only way to create Associations (Triples) and add them to the associative store. If the association already exists in the store, no alterations are made. The argument to the Make statement is a triple of item expressions:

```
MAKE item1 ⊗ item2 ≡ item3
MAKE item1 ⊗ itemvarl ≡ NEW
MAKE itemvar_array[23] ⊗ item1 ≡ itemvar2
```

The component item expressions are evaluated left to right. The three items that the three expressions evaluate to are then formed into an association, and the association is hashed into the associative store. The item expressions must be constructive, that is, one may use the NEW function but not the ANY or BINDIT items (see NEW, page 64, ANY, page 64, and BINDIT, page 65).

BRACKETED TRIPLES ITEMS
Items may be created by declaration, by the NEW function, or by using BRACKETED TRIPLES in Make statements. A Bracketed Triple item may not have a datum, but may have a prop or a pname (see page 84 for pnames, page 56 for props). Instead, a Bracketed Triple items has an Association connected to it. One creates a Bracketed Triple item by executing a Make statement:

```
MAKE item1 ⊗ [item2⊗item3≡item4] ≡ item5
```

where the itemN are item expressions. "[item2⊗item3≡item4]" is the Bracketed Triple item, and of course need not always be the second component of the association. The association connected to the Bracketed Triple item is "item2 ⊗ item3 ≡ item4". The above Make statement actually creates two triples and one item. Namely, the associations

```
item1 ⊗ itemXX ≡ item5
item2 ⊗ item3 ≡ item4
```

and the item "itemXX" which is a Bracketed Triple item and has the second association connected to it. One can access a Bracket Triple item, with the an associative search called the Bracketed Triple Item Retrieval:

```
itmvar ← [itm2 ⊗ itm3 ≡ itm4];
    COMMENT itmvar now contains itmXX;
```

The Bracket Triple construct may be used in any expression See page 58.

Having "itmXX", one may access the items of the association connected to with the predeclared functions FIRST, SECOND, and THIRD (see page 84, for more information on these runtime functions):

```
FIRST ( itemXX ) is item2
SECOND ( itemXX ) is item3
THIRD ( itemXX ) is item4
```

ERASE
The way to remove an association from the associative store and destroy it is to ERASE it:

```
ERASE item1 ⊗ item2 ≡ item3
```

where the itemN are item expressions. The item expressions must be retrieval item expressions that is, one may use the ANY item but not the NEW function or the BINDIT item (see ANY, page 64, and NEW, page 64, and BINDIT page 65). Using ANY as one, two, or three of the item expressions allows many associations to be erased in one statement. If the association to be erased does not exist, Erase is a no-op.

Whenever one Erases an association, none of the items of the association are deleted. In particular, when one Erases an association that has a Bracketed Triple item as one of its components, the Bracketed Triple item is not deleted. Furthermore, the association connected to the Bracketed Triple item is not automatically erased by erasing an association containing a Bracketed Triple item. The following Erase erases only one association:

```
ERASE item1 ⊗ [item2⊗item3≡item4] ≡ item5
```

However, erasing the association connected to a Bracketed Triple deletes the item. Deleting the Bracketed Triple item DOES NOT erase the association connected to it.

11.4 - SEARCHING THE ASSOCIATIVE STORE

Flexible searching and retrieval are the main motivations for using an associative store. It follows that this is the most important section of the Leap part of this manual. It is a rare Leap program that does not use at least one of the searches described below.

Four methods of searching the associative exist in Sail:

Binding Booleans,
Derived Sets,
Bracketed Triple item retrieval, and
Foreach Statements

The first three are properly part of the discussion of Leap Expressions in the next chapter, but are included here for completeness.

Throughout this section we will use the following notat ion for an association:

A⊗O≡V

where A, O and V stand for the "attribute", "object" and "value" items of an association.

The terms "bound" and "unbound" will find heavy use in this section. Bound describes an itemvar that has an item assigned to it. Unbound describes an itemvar that, at this time in the execution of the program, has no item bound to it. The object of searching the associative store is usually to bind unbound itemvars to specific, but unknown, items. If the itemvar to be bound was declared Checked, then type checking will be done, and the appropriate error message will be issue if the binding item does not have the same type as the itemvar.

Throughout this section, references to item expressions will always mean retrieval item expressions. Don't use NEW in such expressions.

A hashing algorithm is used in storing and retrieving associations in Leap. The user can increase the speed of associative searching or decrease his core image by using the REQUIRE n BUCKETS construct to control the size of his associative search hash table to reflect the number of assciations he will be using. A hash table will be allocated with (2↑m) hash codes where m is the smallest integer such that (2↑m) ≥ n. Sail initializes the hash size to '1000.

BINDING BOOLEANS

A Binding Boolean searches the associative store for a specified triple, returning true if one can be found, and false otherwise. A Binding Boolean is a triple:

```
itml ⊗ itm2 ≡ itm3
```

where "itmN" is one of three things: an item expression, or the reserved word "BIND" followed by an itemvar, or the token "?" followed by an itemvar. An item expression as a component of the Binding Boolean means that component of the triple that the boolean finds must be the item specified by the item expressron (unless the item expression evaluates to the item ANY, which specifies that any item is okay). If a "BIND" itemvar is the A, 0 or V of the triple, then the Binding Boolean will attempt to find an association which meets the constraints imposed by the item expression A, 0 or V components, and then binds to the "BIND" itemvar the items occuring in the corresponding positions of the association that the Binding Boolean found. If no such association can be found, then the Binding Boolean returns FALSE and leaves the "BIND" itemvars with their previous values. If "?" precedes an itemvar, then the itemvar will behave like a "BIND" itemvar if it is currently contains BINDIT, but will behave like an item expression if it is bound to some other item than BINDIT. Example:

```
IF Fathr ⊗ ?Son ≡ ANY THEN PUT Son IN Sonset;
IF -Father ⊗ BIND Son ≡ Bob THEN CHILDLESS(Bob);
ERCHEK ← Fathr ⊗ COP(Sonset) ≡ ANY;
```

DERIVED SETS

Derived Sets are quite simple: "Foo ⊗ Garp" where Foo and Garp are item expressions, is the set of all items X such that Foo ⊗ Garp ≡ X exists. "Garp ≡ Sister" is the set of all items X such that X ⊗ Garp ≡ Sister exists. "Foo ' Sister" is the set of all items X such that Foo ⊗ X ≡ Sister exists. Examples:

```
Dadset ← Fathr ⊗ ANY;
Danson ← Fathr ' Dan;
News ← (Son ≡ Dad) ∩ attset;
```

ANY specifies "I don't care" to the search. BINDIT has no special meaning to the search, and behaves like any other items. Since BINDIT can never appear in an association, this means the set returned will always be the empty set PHI.

BRACKETED TRIPLE ITEM RETRIEVAL

A Bracketed Triple item can be referenced by specif ing the association it is connected to. For example,

```
Itmvar ← [itml ⊗ itm2 ≡ ANY]
PUT [ANY ⊗ ANY ≡ ANY] IN Bracset
IF Foo ⊗ Garp ≡ [itml ⊗ itm2 ≡ ANY] THEN . . .
Itmvar ← [itml ⊗ [itm2 ⊗ itm3 ≡ itm4] ≡ itm5]
```

where itmN is any item expression not containing NEW or BINDIT. ANY means you don't care what item occupies that component. If the designated Bracketed Triple is not found, an error message is given.

THE FOREACH STATEMENT

This statement is the heart of Leap. It is similar to the FOR statement of Algol in that a statement is executed once for each binding of a variable. In this semi-schematic example,

```
FOREACH X SUCH THAT <element> AND . . . AND
   <element> DO <statement>;
```

the <statement> is executed once for each binding of the itemvar X. The <element>s in the element list (i.e. <element> AND...AND <element>) determine the bindings of the itemvar, and hence how many times the <statement> is executed. If the <element>s are such that there is no binding possible for X, then the

<statement> is never executed. Like a Sail FOR statement, one may use DONE, NEXT, and CONTINUE within the <statement>. As before, when one uses a NEXT inside the loop, the word NEEDNEXT must precede the FOREACH of the Foreach that one wants checked and possibly terminated. See pages 17, 17, and 18 for more information about Done, Next, and Continue.

Restriction: Jumping (i.e. with a GO TO) into a Foreach is illegal However, it is legal to jump out of a Foreach, or to jump around within the same Foreach.

Foreach statements differ from For statements in that more than one itemvar may be included to be given bindtngs:

```
FOREACH X, Y, Z SUCH THAT <element>. . . .
```

X, Y, and Z are called Foreach itemvars. Just as one must declare the integer I before using it in the Sail For statement

```
FOR I ← 1 STEP 2 UNTIL 21 DO.. .
```

so must one declare Foreach itemvars before using them in Foreaches. Foreach itemvars are no more than normal itemvars receiving special assignments; they may have any type. If a Foreach itemvar that has been declared Checked is assigned an item by the search that has a different type than the Checked itemvar, an error message will result.

Foreach itemvars differ from For variables in a more radical way It is possible to specify to the Foreach that a certain Foreach itemvar be a variable to the search only on the condition that that the itemvar contains the special item BINDIT at the time the Foreach is called. One precedes such itemvars with the "?" token. For example:

```
FOREACH ?X, ? Y, Z SUCH THAT <element>. , , .
```

If X contains BINDIT but Y does not when this Foreach starts execution, then the search will be conducted exactly as if the statement

```
FOREACH X,Z SUCH THAT <element>. . . .
```

were the Foreach specified. The itemvar X will then act just like an ordinary, non-foreach itemvar that was bound previous to the Foreach. All Foreach itemvars may be "?" itemvars if this is desired.

There are four different types of <element> that may be used in foreach element lists:

Set Membership,
Boolean Expressions,
Retrieval Triples, and
Matching Procedures.

The order of the <element>s in the element list is very important, as we shall see.

Terminology: we say that a certain binding of the the Foreach itemvars "satisfies" an <element>. If that binding satisfies each <element> of the element list, then we say it "satisfies the associative context". A fancy way of refering to the element list is "associative context". We also refer to the collection of bindings that satisfy the associative context as the "satisfier group" of the Foreach.

The execution of a Foreach proceeds as follows. After initialization, the Foreach proceeds with a search specified by the first <element> of the element list. If a binding can be found that satisfies the first <element>, the Foreach proceeds forward to the new <element> of the list and trys to satisfy it, and so on. When the Foreach can not satisfy an <element>, it "backs up" to the previous element and trys to get a different binding. If it can't find satisfaction there, it backs up again and trys again to get a different binding. When a Foreach proceeds forward off the end of the element list (i.e. the associative context is satisfied) then the <statement> is executed, and the Foreach backs up to the last <element> of the element list. When the Foreach backs up off the left end of the element list, the Foreach is exited.

When a Foreach is exited by backing up off the left, the Foreach itemvars are restored to the last satisfier group bound to them, regardless of what the <statement> may have done. If the associative context was never satisfied, then the Foreach itemvars have the values that they had before the Foreach. When a Foreach is exited with a GO TO, DONE, or RETURN, the Foreach leave the itemvars with the bindings they had at the GO TO, or whatever, including any modifications that the <statement> may have made to them.

THE LIST MEMBERSHIP <ELEMENT>

[In the following, one may also read "set" for "list"; Sail automatically coerces set expressions into list expressions1 This <element> does not search the associative store to bind an itemvar, but merely binds it with an item of a specified list. In the Foreach,

```
FOREACH X | X IN L DO <statement>;
```

(here we have used the Sail synonym "|" for "SUCH THAT"), the Foreach itemvar X is bound successively

to each element of the set L, starting at the beginning of the list. If an item occurs n times in L, then X will be bound to that item n times in the course of the For-each. Thus, the number of satisfiers to the above Foreach is LENGTH(L).

In the current implementation of Leap, there is a difficulty that should be pointed out. If inside the <statement>, one changes L by list assignment, Removes, etc. in such a way as to remove the next item of the list that the Foreach itemvar would have been bound to, Leap may go crazy. Foreach searches look one ahead and save a pointer to the next items to be bound to the Foreach itemvars. This allows one to remove the items of the current bindings of the Foreach itemvars from lists or whatever, but makes other removals hazardous. For example,

```
FOREACH X | X IN L DO REMOVE X FROM L;
```

will work, but

```
PUT V IN L BEFORE FOO;
FOREACH X | X IN L DO REMOVE V FROM L;
```

will probably fail. No error checking is done.

Whenever the Foreach itemvar of a list <element> has been bound previously, the list element behaves like a boolean. It does not rebind the itemvar but only checks to see that it is in the list. For example,

```
FOREACH X | X IN L AND X IN LL DO <statement>;
```

X is bound by the <element> "X IN L". <element> "X IN LL" is satisfied if the item contained in the itemvar X is in the list LL.

If two different Foreach itemvars are used with two different lists, i.e.

```
FOREACH X,Y | X IN L AND Y IN LL
DO <statement>;
```

then after execution of the <statement>, the Foreach will go back the last <element> that searches for bindings, in this case "Y IN LL" and gets a new binding for Y. It is only on failure of this search that the Foreach goes back to the first <element>, "X IN S", and gets a new binding for X. Thus the <statement> will be executed once for each possible X,Y pair. In the Foreach,

```
FOREACH X,Y | X IN L AND Y IN L...;
```

X and Y will be bound to all possible pairs of elements in L. This includes pairs with duplicate elements, like (a,a). Different orderings of the same elements will NOT be Ignored. Thus, pairs like (a,b) and (b,a) will each be a satisfier group sometime during the Foreach. Furthermore, if the list L contains duplications of the same item, identical pairs will occur in proportion to the number of duplications. That is, regardless of the duplications within the list, the number of satisfier groups to the Foreach above is LENGTH(L)↑2.

THE BOOLEAN EXPRESSION <ELEMENT>
Any Sail boolean expression may be used as an <element> in the Associative Context of a Foreach if it is inclosed by parentheses. A Boolean Expression <element> is satisfied if it is TRUE. Note that the boolean expression must have parentheses around it.

WARNING: Foreach itemvars can not be bound by a Boolean Expression <element>. Therefore, all itemvars used in a Boolean Expression <element> must be bound by previous <element>s in the element list. A Boolean Expression <element> with unbound Foreach itemvars in it causes an error message.

THE RETRIEVAL TRIPLE <ELEMENT>
To search the associative store with a Foreach, one uses the Retrieval Triple <element>. A Retrieval Triple is satisfied if a binding of the Foreach itemvars can be found such that the triple is an extant association, If all of the itemvars of the Retrieval Triple <element> were bound previous to the execution of the Retrieval Triple <element>, then the Triple does no further binding; it is satisfied if the specified triple is in the associative store. For example,

```
FOREACH X | FATHER ⊗ TOM ≡ X AND
   X IN PTA-SET DO <statement>;

FOREACH X | X IN PTA-SET AND
   FATHER ⊗ TOM ≡ X DO <statement>;
```

The two Foreaches have the same effect. However, in the first case, X is bound by a search of the associative store for any triple that has FATHER as its attribute component, and TOM as its object component, When such a triple is found, X is bound to the item that is the value component. Then, if X is in the PTA-SET, the Foreach lets the statement execute. If X is not in PTA-SET, then the Foreach backs up and trys to find another triple with FATHER as its attribute and TOM as its value. In the second Foreach, X is bound with an item from PTA-SET, then the associative store is checked to see that the triple FATHER⊗TOM≡x, where x is the binding of X, is in the store. If it is, the <statement> is executed, otherwise the Foreach backs up and gets a different item from PTA-SET and binds that to X. Assuming that Tom has only one father, the first search is much faster.

Using ANY in a Retrieval Triple indicated that you don't cat-e what item occupies that position. For Instance, In

```
FOREACH X | FATHER ⊗ ANY ≡ X DO <statement>;
```

X is bound successivly to all fathers. However, if the associative store included the following three associations,

```
FATHER ⊗ KAREN ≡ PAUL
FATHER ⊗ LYNN ≡ PAUL
FATHER ⊗ TERRY ≡ PAUL
```

then X would be bound to PAUL only once, not thrice. BINDIT has no special meaning to the search. Since BINDIT can never appear in an association, a Retrieval Triple containing it will cause the search to always fail.

Different kinds of associative searches proceed with different efficiencies. Listed below in order of decreasing efficiency are the various forms of Retrieval Triple <element>s that are legal. A, 0, and V represent either--bound Foreach itemvars or items from explicit item expressions in the triple. x, y, and z represent unbound Foreach itemvars or the item ANY. (note that x ⊗ x ≡ V is really x ⊗ 0 ≡ V, and so on). The two forms of the List Membership <element> are included for comparison.

x IN L	All items x in the list L.
A ⊗ O ≡ x	Only the value is free.
x ⊗ y ≡ V	Attribute and object are free.
A IN L	Verification that item A is in list L.
A ⊗ O ≡ V	Verification that the triple is in the store.
A ⊗ x ≡ V	Only the object is free.
x ⊗ O ≡ V	Only the attribute is free.
A ⊗ x ≡ y	Object and value are free.
x ⊗ O ≡ y	Attribute and value are free.
x ⊗ y ≡ z	Attribute, value and object are free.

Note that MAKEing an association inside a Foreach may or may not affect subsequent bindings. For example, in

```
FOREACH X,Y | Link ⊗ X ≡ Y DO
        MAKE Link ⊗ X ≡ Newlink;
```

it is uncertain whether Y will ever receive Newlink as its binding or not.

The A, 0, and V used in a Retrieval Triple of a Foreach may be a derived set expressions as well as item expressions. For example,

```
FOREACH X, Y | Link ⊗(Father⊗Y) ≡ X DO . . .;
```

ERASE in the <statement> of a Foreach that binds any of its itemvars with Retrieval Triples may cause problems. This is similar to REMOVE used in Foreaches with List Membership <element>s controling some bindings. ERASE can only be guaranteed to to work safely if the association erased is the one we just got a binding from, e.g.

```
FOREACH X | A ⊗ 0 ≡ X DO ERASE A ⊗ 0 ≡ X;
```

or if the association erased could not possible be used for a binding of a Foreach itemvar, such as,

```
FOREACH X | Link ⊗ X ≡ Node DO
        ERASE Node ⊗ X ≡ ANY;
```

Foreaches look one ahead to the next binding of its itemvars, and leaves a pointer to those associations. If you Erase any of those associations, the Foreach gets lost in the boondocks. No error checking is done.

However, as long as the associative store is not changed during the execution of the Foreach, a Retrieval Triple will not itself repeat a particular set of bindings that it bound before.

THE MATCHING PROCEDURE <ELEMENT>

Matching Procedures are the most general search mechanism in Leap. They also provide a convenient method of writing coroutines.

A MATCHING Procedure is very similar to a boolean procedure (in fact outside of Foreach associative contexts, it behaves like a boolean procedure and may be called within expressions, etc.). They must be declared of type MATCHING. They may not be declared SIMPLE. The formal parameters of a Matching Procedure may include zero or more "?" itemvars (pronounced "question itemvars") which may have any datum type but may not be VALUE or REFERENCE. These parameters correspond roughly to either call by value or call by reference, depending on the actual parameter when the procedure is called. When the actual parameter is an item expression or a bound itemvar the parameter is equivalent to a value parameter. However, if the actual parameter is an unbound Foreach itemvar, then the parameter is treated as a reference parameter, and on entry is is initialized to the special item BINDIT.

Matching Procedures are exited by SUCCEED and FAIL statements instead of RETURN statements. When used outside of an associative context, SUCCEED corresponds to RETURN(TRUE) and FAIL corresponds to RETURN(FALSE) [this is not strictly true when the matching procedure is sprouted as a process -- see page 691. Inside an associative context, Succeed and Fail determine whether the Foreach is to proceed to the next <element> of the element list or to backup to the previous <element> of the element list. When

the Foreach backs up into a Matching Procedure, the procedure is not recalled, but resumed at the statement following the last Succeed executed. On the other hand, when a Foreach procedes forward into a Matching Procedure, the procedure is called, not resumed. Therefore, a Matching Procedure <element> will never be resumed following a FAIL statement.

When a Matching Procedure is the last <element> of the associative context, Succeeding will cause the <statement> to be executed; the Foreach then backs up into the Matching Procedure, and the Matching Procedure is resumed at the statement following the Succeed. When a Matching Procedure is the first <element> of an associative context, Failing will exit the Foreach.

WARNING: Matching procedures are actually implemented as processes and therefore two calls of the same matching procedure may share the same memory (see Memory Accessible to a Process, page 68). For example, two calls of the same matching procedure inside the same Foreach (one may even be in the <statement> of the Foreach) will normally share the same memory locations for their locals. To give separate matching procedure calls separate memory locations for their locals, declare the matching procedure RECURSIVE.

When a Matching Procedure is used exterior to the associative context of a Foreach, one may use "BIND" in the call preceding those actuals which one wishes bound regardless of their current binding. Preceding the actual with "?" will have the save effect as "BIND" if the current value of the itemvar is BINDIT, and will have no effect otherwise (the procedure will not attempt to find it a binding)

That is all there is to Matching Procedures. Their power lies in the using them cleverly. The following program illustrates techniques one may use with matching procedures by simulating the List Membership and Retrieval Triple <element>s with matching procedures.

```
RECURSIVE MATCHING PROCEDURE INLIST(? ITEMVAR X; LIST L);
BEGIN "INLIST"
    COMMENT THIS PROCEDURE SIMULATES THE CONSTRUCT
        X ∈ L FOR ALL CASES EXCEPT THE SIMPLE
        PREDICATE BINDIT ∈ L;

        IF X ≠ BINDIT THEN
           BEGIN WHILE LENGTH(L) DO
                      IF X = LOP(L) THEN
                         BEGIN SUCCEED; DONE; END;
                FAIL;
           END;
        WHILE LENGTH(L) DO
           BEGIN X←LOP(L);
                SUCCEED;
           END;
END "INLIST";

MATCHING PROCEDURE TRIPLE(? ITEMVAR A,O,V);
BEGIN "TRIPLE"
DEFINE BINDING(A)="(A=BINDIT)";
SET SET1; INTEGER INDX;
RECURSIVE PROCEDURE SUCC_SET(REFERENCE
                    ITEMVAR &SET S1);
    WHILE LENGTH(S1) DO
    BEGIN X←LOP(S1);
          SUCCEED;
    END;

INDX ← 0;
IF BINDING(A) THEN INDX ← 1;
IF BINDING(O) THEN INDX ← INDX t 2;
IF BINDING(V) THEN INDX ← INDX t4;
CASE INDX OF
BEGIN [0] "A⊗O≡V" IF A⊗O≡V THEN SUCCEED;
      [1] "?⊗O≡V" SUCC_SET(A,O≡V);
      [2] "A⊗?≡V" SUCC_SET(O,A'V);
      [3] "?⊗?≡V"
            BEGIN SET1 ← ANY ≡ V;
                  WHILE (LENGTH(SET1)) DO
                  BEGIN A ←LOP(SET1);
                        SUCC_SET(O,A'V)
                  END;
            END;
      [4] "A⊗O≡?" SUCC_SET(V,A⊗V);
      [5] "?⊗O≡?"
            BEGIN SET1 ← 0 ≡ ANY;
                  WHILE (LENGTH( SET1 )) DO
                  BEGIN A ←LOP(SET1);
                        SUCC_SET(V,A⊗O);
                  END;
            END;
      [6] "A⊗?≡?"
            BEGIN SET1 ← A ' ANY;
                  WHILE (LENGTH(SET1)) DO
                    BEGIN 0 ←LOP(SET1);
                          SUCC_SET(V,A⊗O);
                    END;
            END;
      [7] "?⊗?≡?"
            USERERR(0,1,"ANY⊗ANY≡ANY IS IN BAD TASTE")
    END;
END "TRIPLE";
```

SECTION 12

LEAP EXPRESSIONS

12.1 - SYNTAX

```
<leap-expression>
    ::= <item-expression>
    ::= <set-expression>
    ::= &t-expression>

<item-expression>
    ::= <item_primary>
    ::= [ <item_primary> ⊗ <item_primary> ≡
         <item_primary> ]

<item_primary>
    ::= NEW
    ::= NEW ( <algebraic-expression> )
    ::= NEW ( <set-expression> )
    ::= NEW ( <list-expression> )
    ::= NEW ( <array-name> )
    ::= ANY
    ::= BINDIT
    ::= <item_identifier>
    ::= <itemvar_variable>
    ::= <list-expression> [
         <algebraic-expression> ]
    ::= <itemvar_procedure_call>
    ::= <resume-construct>
    ::= <interrogate-construct>

<itemvar_procedure_call>
    ::= <procedure-call>

<list-expression>
    ::= <list-primary>
    ::= <list-expression> & <list-expression>

<list_primary>
    ::= NIL
    ::= <list-variable>
    ::= {{ <item_expr_list> }}
    ::= ( <list-expression> )
    ::= <list-primary> [ <substring_spec> ]
    ::= <set_primary>

<item_expr_list>
    ::= <item-expression>
    ::= <item_expr_list> , <item-expression>

<set-expression>
    ::= <set_term>
    ::= <set-expression> ∪ <set_term>

<set_term>
    ::= <set-factor>
    ::= <set-term> ∩ <set_factor>

<set-factor>
    ::= <set_primary>
    ::= <set-factor> - <set_primary>

<set_primary>
    ::= PHI
    ::= <set-variable>
    ::= {item_expr_list}
    ::= ( <set_expression> )
    ::= <derived-set>

<derived-set>
    ::= <item-expression> <associative-operator>
         <item-expression>

<associative-operator>
    ::= ⊗
    ::= '
    ::= ≡

<itemvar_variable>
    ::= <variable>

<set-variable>
    ::= <variable>

<list-variable>
    ::= <variable>

<leap-relational>
    ::= <item-expression> IN
         <set-expression>
```

::= <item-expression> IN
<list-expression>
::= <item-expression>
<item_relational_operator>
<item-expression>
::= <set-expression>
<set-relational-operator>
<set-expression>
::= <list-expression>
<list_relational_operator>
<list-expression>
::= <triple>

<item_relational_operator>
::= =
::= ≠

<set-relational-operator>
::= =
::= ≠
::= <
::= >
::= ≤
::= ≤

<list_relational_operatior>
::= =
::= ≠

12.2 - SEMANTICS

ITEM EXPRESSIONS
Itemvars and itemvar arrays may be used in item expressions just as algebraic variables and algebraic arrays are used in algebraic expressions. Itemvars and itemvar arrays are initialized to the special Sail item ANY.

Items may be retrieved from sets and lists with the Sail functions COP and LOP. COP(<set expression or list expression>) yields the item which is the first element of the set or list that the set or list expression evaluated to. LOP also yields the first item of the set or list, but removes that item from the set or list. Because LOP changes the contents of the set or list that is its argument, it can only accept set or list variables, not expressions. See page 41.

List element designators may be used as itemvars in expressions. For example, if RECORD is a list, and ITMVR an itemvar,

```
RECORD[5]← ITMVR;
ITMVR ←RECORD[∞-1];
RECORD[∞]←RECORD[1];
```

are all legal. The special token "∞" means the length of the list when used in this context. The contents of the square brackets may be any algebraic expression as long as it evaluates to an integer n where 1 ≤ n ≤ LENGTH(list).

<list_expression> [<algebraic_expression>] returns a particular element of a list, but may not appear on the left of an assignment expression, because assignment must be to variables.

NEW
The function NEW creates an item at execution time. Since space must be allocated at loading for various tables, one must indicate approximately how may NEW items he will create (the compiler counts the declared items for you). Therefore, one should say "REQUIRE n NEW-ITEMS" where n is some integer less than 4090 (the maximum number of items allowed in Sail). n may be larger than the actual number of New items created, but the excess will be wasted space. If 0 < n < 50, you get tables for 50 New items anyhow.

NEW may take an argument. In this case, the datum of the created item is preloaded with the value passed as argument. If this argument is algebraic, set or list, then the datum will be of the same type. No type conversions are done when passing the algebraic argument. NEW will also accept an array name as argument. In this case, the created item will be of the type array. In fact, the array cited as argument will be copied into the newly created array. The new array will have the same bounds and number of dimensions as the array cited as argument. This array will not disappear even if the block that the original array was declared in is exited. It will only be deallocated if the item is deleted.

NEW in an item expression makes that item expression a "constructive item expression". Constructive item expressions are illegal in some places, namely anywhere that attempts to gets an item from an existing structure (i.e. ERASE, REMOVE, and Associative searches). It is usually clear whether or not a constructive item expression is illegal.

ANY
Some associative searches may need only partial specification. The ANY item is used to specify exactly which parts of the specification are "don't cares"'s. Examples:

```
FOREACH X SUCH THAT Father ⊗ X ≡ ANY DO . . .
IF Father ⊗ BIND X ≡ ANY THEN . . .
```

ANY in an item expression makes that item expression a "retrieval item expression". This is the opposite of a constructive item expression, and is illegal anywhere the statement is creating new structure, namely, a MAKE statement. Thus, ANY is legal everywhere items are, except a MAKE statement.

BINDIT
Like ANY, BINDIT specifies no constraints on the associative search. However, BINDIT has a special meaning to some searches, namely the Binding Boolean and Matching Procedures (depending on how they're written). An itemvar containing BINDIT will be bound by the search to an item of the association that the search found. For example:

```
X ←BINDIT;
IF Father ⊗ ? X ≡ Bob THEN PUT X IN Bobfatherset;
```

Like ANY, BINDIT is illegal in MAKE statements. In certain associative searches, namely the ERASE statement, the Bracketed Triple Item retrieval expression, and the Retrieval Triple <element> of a Foreach, inclusion of BINDIT will cause the search to always fail, because BINDIT can appear in no association.

TYPES AGAIN
The compiler can determine the type of items when the item expression is a typed itemvar, a typed itemvar procedure, a declared item with a type, a typed itemvar array, or a NEW with an argument. When the compiler can determine the type of the item expression, then and only then is it legal to use the Datum construct on the item expression or to assign the item expression to a Checked itemvar. For example, the following are ILLEGAL:

```
DATUM(COP( <set> ))
DATUM(RECORD[∞]);    COMMENT RECORD is a list;
CHEC ← NEW; COMMENT CHEC is a Checked itemvar;
```

SET AND LIST EXPRESSIONS
Three rather standard operations are implemented for use with sets. These are union (∪), intersection (∩), and subtraction (-). These operators have the standard mathematical interpretations. The only possible confusion pertains to subtractions: if we perform the set operation

```
set1 - set2
```

and if there is an instance of an item x in set2 but not in set 1, the subtraction proceeds and no error message is given.

If one considers a list to be a string of items, then concatenation and taking sublists suggest themselves as likely list operations. The syntax and semantics for sublisting and list concatenation are identical with those of strings, with the natural exception that the results are lists, and not strings. There is also a difference in that if the indices to the substringer do not make sense, an error message is generated rather than setting of the _SKIP_ variable. Examples:

```
LISTVAR ←LISTVAR[2 TO ∞-1];
LISTVAR ←LISTVAR[9 FOR 2*N];
LISTVAR ←LISTVAR[1 FOR 2] & LISTVAR[3 TO ∞];
```

One may generate sets with

```
{item1,item2,item3}
```

and may generate lists with

```
{{item1,item1,item2,item3}}.
```

Sets are initialized to the empty set, PHI. Lists are initialized to the null list, NIL. Initialization occurs at the beginning of the execution of the program. Sets and list are reinitialized on entering the blocks of their declaration only when such blocks are in recursive procedures.

DERIVED SETS
Derived sets are really sets of answers to questions which search the associative memory. The conventions are:

```
a ⊗ b -- the set of all x such that a ⊗ b ≡ x
a ≡ b -- the set of all x such that x ⊗ a ≡ b
a ' b -- the set of all x such that a ⊗ x ≡ b
```

BOOLEANS
Several boolean primaries are implemented for comparing sets, lists, and items. In the following discussion, "ix" means item expression, "se" means set expressions, and "le" means list expression. These are:

1) Set and List Membership. The boolean "ix IN se" evaluates the set or list expression, and returns TRUE if the item value specified by the item expression is a member of the set or list.

2) Association Existence. The binding boolean "ix ⊗ ix ≡ ix", where the ix are item expressions or itemvars preceded by ? or BIND, returns TRUE if a binding of the BIND itemvars (and ? itemvars that contained BINDIT) can be found such that the association exists in the associative store. See page 58 for more information on binding booleans.

3) Relations:

ix = ix -- obvious interpretation
ix ≠ ix -- obvious interpretation
se1 < se2 -- true if se1 is a proper subset of se2
se1 ≤ se2 -- true if se1 is identical to se2 or is a proper subset of se2
se1 = se2 -- obvious interpretation
se1 ≠ se2 -- obvious interpretation
se1 > se2 -- equivalent to se2 < se1
se1 ≥ se2 -- equivalent to se2 ≤ se1
le1 = le2 -- obvious interpretation
le1 ≠ le2 -- obvious interpretation

PNAMES
For those desire them, each item may have a string, called its PNAME, linked with it. This is completely independent of the Datum construct. New items and Bracketed Triple items are created with NULL strings as their Pnames. One may delete an item's Pname with the DEL-PNAME function which takes an item expression as its argument. One may give a Pnameless item a Pname with the NEW-PNAME procedure, which takes an item expression and a string as its arguments. CVSI will give you the Pname of an item, and CVIS with give you the item with the specified Pname. No two items may have the same Pname. Pnames do not follow Algol scope rules. See page 84 to find out how to use the above four functions.

If you would like your declared items to have Pnames that are the same as the identifier used in their declaration, say "REQUIRE PNAMES" or "REQUIRE n PNAMES" before their declaration at the beginning of the program. The n is an estimate of the number of dynamically created items with pnames you will use -- this causes tables for n pnames to be allocated at compile time rather than runtime, thus making your program more efficient.

PROPS
Any item may have a PROPS. This is an extra 12 bits of storage (frequently used for bits). PROPS (X) where X is an item expression is exactly an integer variable in its syntax. See page 56 for futher Information on props

SECTION 13

PROCESSES

13.1 - INTRODUCTION

A PROCESS is a procedure call that may be run independently of the main program. Several processes may "run" concurrently. When dealing with a multi-process system, it is not quite correct to speak of "the main program". The main program is actually a process itself, the main process.

This section will deal with the creation, control, and destruction of processes, as well as define the memory accessible to a process. The following section will describe communication between processes

13.2 - SYNTAX

```
<process-statement>
    ::= <sprout-statement>
    ::= <terminate-statement>
    ::= <suspend-statement>
    ::= <join-statement>

<sprout-statement>
    ::= SPROUT ( <item-expression> ,
         <procedure-call> ,
         <algebraic-expression> )
    ::= SPROUT ( <item-expression> ,
         <procedure-call> )

<terminate-statement>
    ::= TERMINATE ( <item-expression> )

<suspend-statement>
    ::= SUSPEND ( <item-expression> )

<join_statement>
    ::= JOIN ( <set-expression> )

<resume-construct>
    ::= RESUME ( <item-expression> ,
         <item-expression> ,
         <algebraic-expression> )
    ::= RESUME ( <item-expression> ,
         <item-expression> )
```

13.3 - SEMANTICS

STATUS OF A PROCESS
A process can be in one of four states: terminated, suspended, ready, or running. A terminated process can never be run again. A suspended process can be run again, but it must be explicitly told to run by some process that is running. Since SAIL is currently implemented on a single processor machine, one cannot really execute two procedures simultaneously. SAIL uses a scheduler to swap processes from ready to running status. A running process is actually executing, while a ready process is one which may be picked by the scheduler to become the running process. The user may retrieve the status of a process with the execution time routine PSTATUS, page 86.

SPROUTING A PROCESS
One creates a process with the SPROUT statement:

```
SPROUT(<item>,<procedure call>,<options>)
SPROUT(<item>,<procedure call>)
```

<item> is a construction item expression (i.e. do not use ANY or BINDIT). Such an item will be called a process item. The item may be of any type; however, its current datum will be writen over by the SPROUT statement, and its type will be changed to "process item" (see TYPEIT, page 83). RESTRICTION: A user must never modify the datum of a process item.

<procedure call> is any procedure call on a regular or recursive procedure, but not a simple procedure. This procedure will be called the process procedure for the new process.

<options> is an integer that may be used to specify special options to the SPROUTer. If <options> is left out, 0 will be used. The different fields of the word are as follows:

BITS	NAME	DESCRIPTION
14-17	QUANTUM(X)	Q ← IF X=0 THEN 4 ELSE 21X; The process will be given a quantum of Q clock ticks, indicating that if the user is using CLKMOD to handle clock interrupts, the process should be run for at most Q clock ticks, before calling the scheduler. (see about CLKMOD, page 79 for details on making processes "time share").
18-21	STRINGSTACK(X)	S ← IF X=0 THEN 16 ELSE X*32; The process will be given S words of string stack.
22-27	PSTACK(X)	P←IF X=Ø THEN 32 ELSE X*32; The process will be given P words of arithmetic stack.
28-31	PRIORITY(X)	P ← IF X=0 THEN 7 ELSE X; The process will be given a priority of P. 0 is the highest priority, and reserved for the SAIL system. 15 is the lowest priority. Priorities determine which ready process the scheduler will next pick to make running.
32	SUSPHIM	If set, suspend the newly sprouted process.
33		Not used at present.
34	SUSPME	If set, suspend the process in which this sprout statement occurs.
35	RUNME	If set, continue to run the process in which this sprout statement occurs,

The names are defined in the file SYS:PROCES.DEF, which one may require as a source file. Options words may be assembled by simple addition, e.g. RUNME + PRIORITY(3) + PSTACK(2).

DEFAULT STATUS: If none of bits 32, 34, or 35 are set, then the process in which the sprout statement occurs will revert to ready status, and the newly sprouted process will become the running process.

The default values of QUANTUM, STRINGSTACK, PSTACK, and PRIORITY are stored in the system variables DEFQNT, DEFSSS, DEFPSS, and DEFPRI respectively. These values may be changed. The variables are declared EXTERNAL INTEGERs in SYS:PROCES.DEF.

MEMORY ACCESSIBLE TO A PROCESS

A process has access to the same global variables as would a "normal" call of the process procedure at the point of the SPROUT statement. For example, suppose you Sprouted a process in the first instantiation of a recursive procedure and immediately suspended it. Then in another instantiation of the procedure, you resumed the process. Since each recursive instantiation of a procedure creates and initializes new instances of its local variables, the process uses the Instances of the recursive procedure's locals that were current at the time of the SPROUT, namely those of the first instantiation.

Sail will give you an error message whenever the global variables of a process are deallocated but the process still exists. Usually, this means that when the block in which the process procedure was declared is exited, the corresponding process must be terminated (one can insure this by using a small Cleanup procedure that will TERMINATE the fated process or JOIN it to the current one -- see about Cleanup, page 9, Terminate, page 69, and Join statements, page 7Ø). When the process procedure has been declared inside a recursive procedure, things become a bit more complex. As mentioned above, the process takes its globals from the context of the Sprout statement. Therefore, it is only in the instantiation of the recursive procedure that executed the Sprout that trouble can occur. For example,

```
RECURSIVE PROCEDURE TENLEVEL( INTEGER I );
BEGIN "TROUBLE"
   PROCEDURE FOO;
   ; COMMENT does nothing;

   IF I=5 THEN SPROUT(NEW,FOO,SUSPHIM);

     COMMENT sprouts FOO on the 5th
     instantiation of TENLEVEL, then
     immediately suspends it;

   IF I<1Ø THEN TENLEVEL(I+1);
   RETURN;

     COMMENT assuming TENLEVEL is called
     with I=Ø, it will do 10 instantiations,
     then come back up;

END "TROUBLE";
```

TENLEVEL will nest 10 deep, then start returning. This means "TROUBLE" will be exited five times will no ill effects, However, when Sail attempts to exit "TROUBLE" a sixth time, it will be exiting a block in which a process was sprouted and declared. It will generate the error message, "Unterminated process dependent on block exited".

The construct DEPENDENT% <block-name>), where <block-name> is a string constant, produces a set of process items. The process items are those of all the

processes which depend on the current instance of the named block -- i.e. all processes whose process procedures obtain their global variables from that block (via the position of the process procedure's declaration, or occasionaly via the location of the Sprout in a nest of recursive procedure instantiations). This construct may be used together with a CLEANUP procedure (see page 9) to avoid having a block exit before all procedures dependent on it have been terminated.

If one Sprouts the same non-recursive procedure more than once (with different process items, of course), the local variables of the procedure are not copied. In other words, "X←5" in process A will store 5 in the same location that "X←10" in process B would store 10. If such sharing of memory is undesirable, declare the process procedure RECURSIVE, and then new instances of the local variables of the procedure will be created with each Sprout involving that procedure. Then "X" in process A will refer to a different memory location than "X" in process B.

SPROUTING MATCHING PROCEDURES

When a matching procedure is the object of a Sprout statement, the FAIL and SUCCEED statements are Interpreted differently than they would be were the matching procedure called in a Foreach or as a regular procedure. FAIL is equivalent to RESUME (CALLER(MYPROC),CVI(Ø)). SUCCEED is equivalent to RESUME (CALLER (MYPROC),CVI(-1)). RESUME is described on page 69, CALLER on page 85, and MY PROC on page 85.

THE TERMINATE STATEMENT

```
TERMINATE ( <process item> )
```

<process item> may be an item expression, but must yield a process item. It is legal to terminate a terminated process.

Termination of a process causes all blocks of the process to be exited.

A terminated process is truly dead. The item may be used over for anything you want, but after you have used it for something else, you may not do a terminate on it.

SUSPENDING A PROCESS

One can suspend a process with a SUSPEND statement, a RESUME Construct, or a JOIN statement. The suspend statement is simply:

```
SUSPEND ( <process item> )
```

All this does is suspend the process named by the process item. As with the terminate statement, <process item> may be an item expression, but must yield a process item. One may suspend a suspended process. Suspending a terminated process will cause an error message. If the process being suspended is the currently running process (i.e. the process suspends itself), then the scheduler will be called to find another process to run.

THE RESUME CONSTRUCT

General coroutine style interactions are facilitated by the RESUME construct.

```
RESUME ( <process item>, <return item>, <options> )
RESUME ( <process item>, <return item> )
```

<process item> may be any item expression which evaluates to a process item of a suspended process. <return item> is any item expression. <options> is an integer expression.

Resume provides a means for one process to restore a suspended process to ready/running status while at the same time communicating an item to the awakened process. It may also specify what its own status should be. It may be used anywhere that an itemvar procedure is syntactically correct. When a process which has suspended itself by means of a resume is subsequently awakened by another resume, the <return item> of the awakening resume is used as the value of the resume that caused the suspension. For example, suppose that process A has suspended itself with the Resume construct:

```
STARTINFO ←RESUME( Z , NEED-TOOL );
```

If later a process B executes the statement,

```
INFOFLAG←RESUME( A , HAMMER )
```

then B will suspend itself and A will become the running process. A's process information will be updated to remember that it was Awakened by B (so than the runtime routine CALLER can work). Finally, A's resume will return the value HAMMER, which will be assigned to STARTINFO. If A had been suspended by a Suspend statement or a Join statement, then the <return item> of B's Resume is ignored.

Note that a process that has been suspended in any manner will run from the point of suspension onward when it is resumed.

<options> is an integer, used to change the effect of the resume on the current process (Me) and the newly resumed process. If <options> is left out, 0 will be used.

BITS	NAME	DESCRIPTION
33-32	READYME	If 33-32 is 1, then the current process will not be suspended, but be made ready.
	KILLME	If 33-32 is 2, then the current process will be terminated.
	IRUN	If 33-32 is 3, then the current process will not be suspended, but be made running. The newly resumed process will be made ready.
34		This should always be zero.
35	NOTNOW	If set, this bit makes the newly resumed process ready instead of running. If 33-32 are not 3, then this bit causes a rescheduling.

DEFAULT: If none of bits 35 to 32 are set, then the current process wili be suspended and the newly resumed process will be made running. Include a REQUIRE "SYS:PROCES.DEF" SOURCE-FILE in your program to get the above bit names defined. Options may then be specified by simple addition, e.g. KILLME + NOTNOW.

THE JOIN STATEMENT

If you have a number of processes running together, you may wish them all to finish. Say:

JOIN(<set expression>)

where <set expression> evaluates to a set containing only process items. The current process (the one with the join statement in it) is suspended until all of the processes in the set are terminated. WARNING: Be very careful with this statement, you can get into infinite wait situations.

1. Do not join to the current process; since the current process is now suspended, it will never terminate of its own accord.

2. Do not suspend any of the joined processes unless you are assured they will be resumed.

3. Do not do an interrogate-wait in any of the processes unless you are sure that the event it is waiting for will be caused (events are explained in sect ion 12).

SCHEDULING

One may change the status of a process between terminated, suspended and ready/running with the TERMINATE, SUSPEND, RESUME, and JOIN constructs discussed above, and the CAUSE and INTERROGATE constructs discussed in the next chapter. This section will describe how the the status of processes may change between ready and running.

Whenever the currently running process performs some action that causes its status to change (to ready, terminated, or suspended) without specifying which process is to be run next, the Saii process scheduler will be invoked. It chooses a process from the pool of ready processes. The process it chooses will be made the next running process. The scheduling algorithm is essentially round robin within priority class. In other words, the scheduler finds the highest priority class that has at least one ready process in it. Each class has a list of processes associated with it, and the scheduler choses the first ready process on the list. This process then becomes the running process and is put on the end of the list. If no processes have ready status, the scheduler looks to see if the program is enabled for any interrupts (see Interrupts, page 78). If the program is enabled for some kind of interrupt that might still happen (not arithmetic overflow, for instance), then the scheduler puts the program in interrupt wait. After the interrupt is dismissed, the scheduler tries again to find a ready process. If no interrupts that may still happen are enabled, and there are no ready processes, the error message "No one to run." is issued.

The rescheduling operation may be explicitly invoked by calling the runtime routine URSCHD, which has no parameters.

POLLING POINTS

Polling points are located at "clean" or "safe" points in the program; points where a process may change from running to ready and back with no bad effects. Polling points cause conditional rescheduling. A polling point is an efficient version of the statement:

```
IF INTRPT ∧¬NOPOLL THEN
    BEGIN INTRPT←0; URSCHD END;
```

INTRPT is an external integer that is used to request rescheduling at the next polling point. It is commonly set by the deferred interrupt routine DFRINT (for all about deferred interrupts, see page 80) and by the clock interrupt routine CLKMOD (for how to make processes time share, see page 79). The user may use INTRPT for his own purposes (carefully, so as not to interfere with DFRINT or CLKMOD) by including the declaration "EXTERNAL INTEGER INTRPT", then assigning INTRPT a non-zero value any time he desires the next polling point to cause rescheduling. NOPOLL

is another external integer that is provided to give the user a means of dynamically inhibiting polling points, For example, suppose one is time sharing using CLKMOD. In one of the processes, a point is reached where it becomes important that the processes not be swapped out until a certain tight loop is finished up. By assigning NOPOLL (which was declared an EXTERNAL INTEGER) a non-zero value, the polling points in the loop are efficiently ignored. Zeroing NOPOLL restores normal time sharing.

A single polling point can be inserted with the statement POLL The construct

REQUIRE n POLLING-INTERVAL

where n is a positive integer, causes polling points to be Inserted at safe points in the code, namely: at the start of every statement provided that at least n instructions have been emitted since the last polling point, after every label, and at the end of every loop. If $n \leq 0$ then no further polling points will be put out until another Require n (n>Ø) Polling-Interval is seen.

SECTION 14

EVENTS

14.1 - SYNTAX

<event-statement>
 ::= <cause-statement>
 ::= <interrupt-statement>

<cause-statement>
 ::= CAUSE (<item-expression> ,
 <item_expression> ,
 <algebraic-expression>)
 ::= CAUSE (<item-expression> ,
 <item-expression>)

<interrogate-construct>
 ::= INTERROGATE (<item-expression> ,
 <algebraic-expression>)
 ::= INTERROGATE (<item-expression>)
 ::= INTERROGATE (<list_expression> ,
 <algebraic-expression>)
 ::= INTERROGATE (-&t-expression>)

14.2 - INTRODUCTION

The Sail event mechanism is really a general message processing system which provides a means by which an occurrence in one process can influence the flow of control in other processes. The mechanism allows the user to classify the messages, or "event notices", into distinct types ("event types") and specify how each type is to be handled.

Any leap item may be used as an event notice. An event type is an item which has been given a special runtime data type and datum by means of the runtime routine:

MKEVTT (et)

where et is any item expression (except ANY or BINDIT). With each such event type Sail associates:

1. a "notice queue" of items which have beeen "caused" for this event type.

2. a "wait queue" of processes which are waiting for an event of this type.

3. procedures for manipulating the queues.

The principle actions associated with the event system are the CAUSE statement and the INTERROGATE construct. Ordinarily these statements cause standard Sail runtime routines to be invoked. However, the user may substitute his own procedures for any event type (see User Defined Cause and Interrogate procedures, page 73). The Cause and Interrogate statements are here described in terms of the SAil system supplied procedures.

14.3 - SAIL DEFINED CAUSE AND INTERROGATE

THE CAUSE STATEMENT

```
CAUSE (<event type>, <event notice>, <options>)
CAUSE (<event type>, <event notice> )
```

<event type> is an item expression, which must yield an event type item. <event notice> is an item expression, and can yield any legal item. <options> is an integer expression. If <options> is left out, 0 is used.

The Cause statement causes the wait queue of <event type> to be examined. If it is non-empty, then the system will give the <event notice> to the first process waiting on the queue (see about the WAIT bit in Interrogate, below). Otherwise, <event notice> will be placed at the end of the notice queue for <event type>.

The effect of Cause may be modified by the appropriate bits being set in the options word:

BITS	NAME	DESCRIPTION
35	DONTSAVE	Never put the <event item> on the notice queue. If there is no process on the wait queue, this makes the cause statement a no-op.
34	TELLALL	Wake all processes waiting for this event. Give them all this

item. The highest priority process will be made running, others will be made ready.

33	RESCHEDULE	Reschedule as soon as possible (i.e. immediately after the cause procedure has completed . executed).

DEFAULT: If bits 35 to 33 are 0, then the either a single process is awakened from the wait queue, or the event is placed on the notice queue. The process doing the Cause continues to run. REQUIRE "SYS:PROCES.DEF" SOURCE-FILE to get the above bit names defined. Options can then be constructed with simple addition, e.g. DONTSAVE t TELLALL.

THE INTERROGATE CONSTRUCT - SIMPLE FORM

<itemvar> ← INTERROGATE (<event type>, <options>)
<itemvar> ← INTERROGATE (<event type>)

<event type> is an item expression, which must yield an event type item. <options> is an integer expression. If <options> is left out, 0 is used.

The notice queue of <event type> is examined. If it is non-empty, then the first element is removed and returned as the value of the Interrogate. Otherwise, the special item BINDIT is returned.

<options> modifies the effect of the interrogate statement as follows:

BITS	NAME	DESCRIPTION
35	RETAIN	Leave the event notice on the notice queue, but still return the notice as the value of the interrogate. If the process goes into a wait state as a result of this interrogate, and is subsequently awakened by a Cause, then the DONTSAVE bit in the Cause statement will override the RETAIN bit in the Interrogate if both are on.
34	WAIT	If the notice queue is empty, then suspend the process executing the interrogate and put its process item on the wait queue.
33	RESCHEDULE	Reschedule as soon as possible (i.e. immediately after execution of the interrogate procedure).
32	SAY-WHICH	Creates the association EVENT-TYPE ⊗ <event notice> ≡ <event type> where <event type> is the type of the event returned. Useful with the set form of the Interrogate construct, below.

DEFAULT: If bits 35 to 32 are 0, then the interrogate removes an event from the event queue, and returns it. If the event queue is empty, BINDIT is returned and no waiting is done; the process continues to run. Use a REQUIRE "SYS:PROCES.DEF" SOURCE-FILE to get the names defined; use simple addition to form options,e.g. RETAIN t WAIT.

THE INTERROGATE CONSTRUCT - SET FORM

<itemvar> ← INTERROGATE (<event type set>)
<itemvar> ← INTERROGATE (<event type set>, <options>)

<event type set> is a set of event type items. <options> is an integer expression. If it is left out, 0 will be used.

The set form of interrogate allows the user to examine a whole set of possible event types. This form of interrogate will first look at the notice queues, in turn, of each event type in <event type set>. If one of these notice queues is non-empty, then the first notice in that queue will be remved and that notice will be returned as the value of the Interrogate. If all the notice queues are empty, and WAITing is not specified in the options word, then BINDIT will be returned. When the WAIT bit is set, the process doing the interrogate gets put at the end of the wait queues of each event type in <event type set>. Then, when a notice is finally available, the process is removed from all of the wait queues before returning the notice. Note that the option SAY-WHICH provides a means for determing which event type produced the returned notice.

14.4 - USER DEFINED **CAUSE AND INTERROGATE**

By executing the appropriate runtime routine, the user can specify that some non-standard action is to be associated with CAUSE or INTERROGATE for a particular event type. Such user specified cause or interrogate procedures may then manipulate the event data structure directly or by themselves invoking the primitives used by the Sail Cause and Interrogate constructs. User defined Cause and Interrogate are not for novice programers (this is an understatement).

EVENT TYPE DATA **STRUCTURE**
The datum of an event type item points to a six word

block of memory. This block contains the following information:

WORD	NAME	TYPE	DESCRIPTION
0	NOTCQ	LIST	The list of all notices pending for this event type.
1	WAITQ	LIST	The list of all processes currently waiting for a notice of this type.
2	---	---	Procedure specifier for the user specified cause procedure (zero if system procedure is to be used).
3	---	---	Procedure specifier for the user specified interrogate procedure (zero if system procedure is to be used).
4	USER 1	INTEGER	Reserved for the user's pleasure.
5	USER2	INTEGER	Reserved for the user's pleasure.

The appropriate macro definitions for these names (e.g. WAITQ(et) = "MEMORY! DATUM(et)+1, LIST]") are included in the file SYS:PROCES.DEF.

USER CAUSE PROCEDURES
A procedure to be used as a Cause procedure must have three formal value parameters corresponding to the event type, event notice, and options of the Cause. Such a procedure is associated with an event type by means of the runtime SETCP:

```
SETCP (<event type>, <procedure specifier>);
```

where <event type> must yield an event type item and <procedure specifier> is either a procedure name or DATUM(<procedure item>).

For example:

```
PROCEDURE CX (ITEMVAR ET, EN; INTEGER OPT);
    BEGIN INTEGER FLAG;
    OUTSTR ("Causing "& CVIS(EN,FLAG) &
    " as an event of type "& CVIS (ET,X));
    CAUSE1 (ET,EN,OPT);
    END;
...
SETCP(FOO,CX);
```

Now,

```
CAUSE (FOO,BAZ);
```

would cause CX(FOO,BAZ) to be called. This procedure would print out "Causing BAZ as an event of type FOO" and then call CAUSE1.

The runtime CAUSE 1(ITEMVAR etype, enot; INTEGER opt) is the SAil runtime routine that does all the actual work of causing a particular notice, enot, as an instance of event type etype. It is essentially this procedure which is replaced by a user specified cause procedure.

CAUSE1 uses an important subroutine which is also available to the user. The integer runtime ANSWER(ITEMVAR ev-type, ev-not, process-item) is used to wake up a process that has suspended itself with an interrogate. If the process named by process-item is suspended, it will be set to ready status and be removed from any wait queues it may be on. ANSWER will return as its value the options bits from the interrogate that caused the process to suspend itself. If the named process was not suspended, then ANSWER returns an integer word with bit 18 (the '400000 bit in the right half = NOJOY in SYS:PROCES.DEF) set to 1. The evfype and ev-not must be included in case the SAY-WHICH bit was on in the interrogate which caused the suspension. ANSWER has no effect on the notice queue of ev-type.

Frequently one may wish to use a cause procedure to re-direct some notices to other event types. For instance:

```
PROCEDURE CXX (ITEMVAR ET, EN; INTEGER OPT);
    BEGIN ITEMVAR OTH; LABEL C;
    IF redirecttest(ET, EN) THEN
        FOREACH OTH | OTHER_CAUSE⊗ET≡OTH DO
            C:    CAUSE1(ET, EN, OPT)
    ELSE CAUSE1 (ET, EN, OPT);
    END;
```

In order to avoid some interesting race conditions, the implementaion will not execute the causes at C immediately. Rather, it will save ET, EN and OPT, then, when the procedure CXX is finally exited, any such deferred causes will be executed in the order in which they were requested.

USER INTERROGATE PROCEDURES
A user specified interrogate procedure must have two value formal parameters corresponding to the two arguments to INTERROGATE and should return an item as the value. The statement

```
SETIP (<event type>, <procedure specifier>);
```

where <event type> is an event type item, and <procedure specifier> is either a procedure name or DATUM (<procedure item>), will make the specified procedure become the new interrogate procedure for <event type>. For instance:

```
ITEMVAR PROCEDURE IX (ITEMVAR ET; INTEGER OPT);
    BEGIN INTEGER FLAG; ITEMVAR NOTI;
    NOTI ← ASKNTC(ET, OPT);
    OUTSTR("Notice " & CVIS(NOTI,FLAG) & " returned
      from interrogation of " & CVIS(ET,FLAG));
    RETURN (NOTI);
    END;
...

SETIP(FOO, IX);
```

Now,

```
    ... ← INTERROGATE(FOO);
```

would cause NOTI to be set to the value of ASKNTC(FOO,Ø) Then the message "Notice BAZ returned from interrogate of FOO" would be printed and IX would return NOTI as its value.

The runtime ASKNTC(ITEMVAR etype; INTEGER opt) is the Sail system routine for handling the interrogation of a single event type. Essentially it is the procedure being replaced by the user interrogate procedure. --

In the case of multiple interrogations, Sail sets a special bit (bit 19 = '200000 in the right half = MULTIN in SYS: PROCES.DEF) in the options word before doing any of the interrogates specified by the event type items in the event type set. The effect of this bit, which will also be set in the options word passed to a user interrogate procedure, is to cause ASKNTC always to return BINDIT instead of ever waiting for an event notice. Then, if ASKNTC returns BINDIT for all event types, Sail will cause the interrogating process to Wait until its request is satisfied. If multin is not set, then ASKNTC will do the WAIT if it is told to.

SECTION 15

PROCEDURE VARIABLES

15.1 - SYNTAX

<assign-statement>
 ::= ASSIGN (<item_expr> ,
 <procedure-name>)
 ::= ASSIGN (<item_expr> ,
 DATUM (<item_expr>))

<ref_item_construct>
 ::= REF_ITEM(<expression>)
 ::= REFJTEM (VALUE <itemvar>)
 ::= REFJTEM (BIND <itemvar>)
 ::= REFJTEM (? <itemvar>)

<apply-construct>
 ::= APPLY (<procedure-name>)
 ::= APPLY (<procedure-name> ,
 <arg_list_specifier>)
 ::= APPLY (DATUM (<item>))
 ::= APPLY (DATUM (<item>) ,
 <arg_list_specifier>)

<arg_list_specifier>
 ::= <list-expression>

15.2 - SEMANTICS

ASSIGN

One may give an item a procedure "datum" using the ASSIGN statement. ASSIGN accepts as its first argument an item expression (do not use ANY or BINDIT). To this is bound the procedure identified by its name or to the "datum" of another procedure item. The procedure may be any type. However, the value it returns will only be accessible if the procedure is an itemvar or item procedure. Apply assumes that whatever the procedure left in AC 1, (the register used by all non-string procedures to return a value) on exiting is an item number. Warning: a procedure is no ordinary datum. Using datum on a procedure item except in the above context will not work. Use APPLY instead.

REF_ITEM

Reference items are created at run time by the REF_ITEM construct and are used principly in argument lists for the APPLY construct. The datum of a reference item contains a pointer to a data object, together with type information abcut that object. To create a reference item one executes

```
itm ← REF,ITEM ( <expression> )
```

A NEW item is created. If the expression is (a) a simple variable or an array element, then the address will be saved in the item's datum. If the expression is (b) a constant or "calculated" expression, then Sail will dynamically allocate a cell into which the value of the expression will be saved, and the address of that ceil will be saved in the datum of the item. The item is then noted as having the datum type "reference" and returned as the value of the REFJTEM construct. One can slightly modify this procedure by using one of the following variations.

```
itm ← REF,ITEM ( VALUE <expression>)
```

In this case, a temp cell will always be allocated. Thus X←3; XI←REF_ITEM(VALUE X); X4-4; would cause the datum of XI to point at a cell containing 3.

```
itm ← REFJTEM ( ? itmvr )
itm ← REF,ITEM ( BIND itmvr )
```

where itmvr must be an itemvar or an element of an itemvar array, will cause the reference item's datum to contain information that Apply can use to obtain the effect of using "? itmvr" or "BIND itmvr" as an actual parameter in a procedure call.

APPLY

APPLY uses the items in the <arg_list_specifier>, together with the environment information from the procedure item (or from the current environment, if the procedure is named explicitly) to make the appropriate procedure call. <arg_list_specifier> is an ordinary list expression, except that each element of the list must be a reference item. The elements of the list will be used as the actuals in the procedure call. There must be at least as many list elements as there are formals in the procedure The reference items must refer to an object of the same type as the corresponding formal parameter in the procedure being called. (EXCEPTION: if the formal parameter is an untyped itemvar or untyped itemvar array, then the reference

item may refer to a typed itemvar or **itemvar array,** respectively). At present, type checking, but not type coercion, is done. If the formal parameter is **a** reference parameter, then a reference to the object pointed to by the reference item **is passed. If the** formal parameter is a value parameter, then **the value** of the object pointed to by the reference **item** is used. Similarly, "?" formals are handled appropriately **when** the reference item contains **a** "?" **or "BIND" reference.** If the procedure to be called **has no parameters, the** <arg_list_specifier> may be left out.

Apply may be used wherever an itemvar procedure call is permitted. The value returned will be whatever value would normally be returned **by the the applied** procedure, but Apply will treat it as an item number. Care should therefore be taken when using the result of Apply when the procedure being invoked is not itself an itemvar procedure, since this **may cause an** invalid item number to be **used as a valid item (for** instance, in a MAKE). Recall that when **a typed** procedure (or an Apply) is called at **statement level,** the value it returns is ignored.

Here is an example of the use of APPLY.

```
BEGIN
LIST L;INTEGER XX;
INTEGER ITEMVAR YY;ITEMVAR ZZ;
REAL ARRAY AA[1:2];
PROCEDURE FOO(INTEGER X;
   ITEMVAR Y,Z; REAL ARRAY A);
      BEGIN
      Y←NEW(X);
      Z←NEW(A);
      A[X]←3;
      END;
XX←0;
L←{{REF_ITEM(XX),REF_ITEM(YY),
   REF_ITEM(ZZ),REF_ITEM(AA)}};
XX←2;AA[1]←AA[2]←1;
APPLY(FOO,L);
COMMENT Y now contains an item whose
   datum is 2, Z contains an item whose
   datum is the array (1.0,1.0),
   A[1]=1.0, and A[2]=3.0.;
END;
```

The variables accessed by a procedure **called with** APPLY may not always be what you would think they were. Temporary terminology: the **"environment" of a** procedure is the collection of variables, **arrays and** procedures accessible to it. "Environment" is not meant to include the state of the **associative store or** the universe of items. The environment of **a** procedure item is the environment of the **ASSIGN, and** that environment will be used **regardless of the** position of the APPLY. Since procedure **items are** untouched by block exits, yet environments **are, it is** possible to Apply a procedure item when its environment is gone; Sail catches most of these situations and gives an error message.

Consider the following example:

```
BEGIN
ITEM P; LABEL L;
RECURSIVE PROCEDURE FOO (INTEGER J);
BEGIN "FOO"
   INTEGER I;
   PROCEDURE BAZ;
     OUTSTR("J="&CVS(J)&" I="&CVS(I));
   IF J=I THEN
      BEGIN
      I←2;
      ASSIGN(P, BAZ);
      FOO(-1);
      END
   ELSE APPLY(DATUM(P));
END "FOO";
FOO(1);
L: APPLY(DATUM(P)); COMMENT will cause a
     runtime error -- see discussion below;
END
```

The effect of the program is to Assign Baz to P on the first instantiation of Foo, then Apply P on the second (recursive) instantiation. However, the environment at the time of the Assign includes {I=2, J=1} but the environment at the time of the Apply includes {I=0, J=-1} instead. At the time of the Apply, Baz is executed with the environment from the time of the Assign, and will print out

```
J=1 I=2
```

The Apply at L will cause a **runtime** error message because the environment of the Assign has been destroyed by the exiting of Foo.

SECTION 16

INTERRUPTS

16.1 - INTRODUCTION

The interrupt facilities of SAIL were built around the user interrupt system provided by the Stnford time sharing system. They will work, in some imited way, for SAILs running on DEC 10-50 systems. In this case, the DEC APRENB trap system is used. This system has a somewhat limited utility when one is dealing with "asynchronous" interrupts (such as the real-time clock), since nothing protects your "interrupt" routine from being itself interrupted. The Stanford SAIL interrupts have been implementer in such a manner that they may be used in prograns that have also enabled themselves for APRENB interrupts (as may happen when one uses various exterral "packages" of procedures). In this case, conditions enabled using the runtime routines described here are processed by the SAIL interrupt handler, and those enabled for APRENB processing are handled by whatever handler the user has provided (an attempt to enable the same condition on both systems causes an error). In export SAIL, the SAIL interrupt handler is directly tied to the APRENB Interrupt system, and thus may not be used with programs that also attempt to enable themselves directly for interrupts.

Essentially, there are two types of interrupt available: immediate and deferred. An immediate irterrupt is executed at the time the condition causing it arises, (usually right after the current instruction finishes -- see [Moorer], II.D.16 for exceptions). A deferred interrupt will we executed at the next "polling point" in the user's program. (See about polling points on page 7Ø).

This chapter will describe both immediate and deferred interrupts and will describe those areas in which the Stanford system differs from the export system (principally: immediate interrupts and the index numbers used to specify interrupt conditions).

16.2 - IMMEDIATE INTERRUPTS

To set up an immediate interrupt, simply say

```
INTMAP( <index>,<simple procedure name>,0);
ENABLE( <index> )
```

where <index> is a code for the interrupt condition (e.g clock, arithmetic overflow, etc.). (The codes, together with the names given them in SYS:PROCES.DEF, may be found in the appendix on Interrupt Codes) The INTMAP statement will inform the SAIL interrupt handler that it is to call the specified procedure (which must be SIMPLE) when it (the interrupt handler) gets invoked for the specified condition. Also, it causes the system user interrupt interface to be set up so that user interrupts are to be sent to the SAIL interrupt handler. The ENABLE statement informs that it is to execute the user interrupt procedure (which was set by INTMAP to be the SAIL interrupt dispatcher) whenever the named condition occurs, An interrupt may be disabled by the statement

```
DISABLE(<index>)
```

The system will not provide user interrupts for the specified condition until another ENABLE statement is executed.

IN STANFORD SAIL
A procedure specified by an INTMAP statement will be executed at a special "user interrupt level". A program operating in this mode will not be interrupted, but must finish whatever it is doing within 1/ 10 th of a second. It may not do any UUOs that can cause it to be rescheduled. Also, the accumulators will not be the same ones as those that were in use by the regular program (ie their values will be different). Certain locations are set up as follows:

ACs l - 6 Set up by the system as in [Moorer] II.D. 16-2

AC '15 (USER) Address of the Sail user table,

AC '16 (SP) A temporary string push down stack pointer. NOTE: extreme care should be used when using strings inside interrupt procedures, since if a string garbage collection should take place or if one was interrupted, then the program will die a terrible death. This means that strings should not be used in any asynchronous interrupt, and that one should avoid doing string concatenations, cvs's. etc.

AC '17 (P) A temporary push down stack pointer.

XJBCNI (declared in SYS:PROCES.DEF as an external integer). Bit mask with **a** bit on corresponding to the current condition.

XJBTPC (declared in SYS:PROCES.DEF as **an** external integer) Full PC word of regular user level program.

The interrupt will be "dismissed", and the user program resumed, when the interrupt procedure is exited. For more information on interrupt level programming, consult the Stanford System documentation.

IN EXPORT SAIL
The interrupt handler again will decode the interrupt condition and call the appropriate procedure. Since there is no "interrupt level", the interrupt procedure must not itself generate any interrupt conditions, since this will cause SAIL to lose track of where in the user program it was interrupted (trapped).

Also, the SAIL interrupt module sets up some temporary accumulators and JOBTPC:

AC '10 index of the interrupt condition.

AC '15 (USER) Address of the SAIL user table

AC '16 (SP) A temporary string push down list. Same warning about the use of strings in Stanford interrupt procedures applies here.

AC '17 (P) A temporary push down pointer

JOBTPC (an external integer) Full PC word of regular user program.

The "real" acs -- ie the values of all accumulators **at** the time the trap occurred -- are stored in locations APRACS to APRACSt17. Thus you can get **at the** value of accumulator x by declaring APRACS **as an** external integer and referring to MEMORY[LOCATION(APRACS)+x]. When the interrupt procedure is exited, the acs are restored from APRACS to APRACS+17, and the SAIL interrupt handler jumps to the location stored in JOBTPC (which was set by the operating system to the location at which the trap occurred). Thus, if **you want to transfer** control to some location in your user program, a good way to do it is to have an interrupt routine like:

```
SIMPLE PROCEDURE IROUT;
  BEGIN
  EXTERNAL INTEGER JOBTPC;

  JOBTPC←LOCATION(GTFOO);
  COMMENT GTFOO is a non-simple procedure
     that contains a GO TO FOO, where FOO
     is the location to which control
     is to be passed. This allows the
     "go to solver" to be called and clean
     up any unwanted procedure activations.;
  END;
```

WARNING: this approach is rather dangerous if the interrupt occurred in certain runtime routines. In particular if you were inside a string **garbage** collection, or allocating an array, you will lose miserably.

THE PROCEDURE CLKMOD
(CLKMOD is currently available only in Stanford Sail) The most common usage of immediate interrupts is to approximate time sharing among processes. Every time the scheduler decides to run a process, it copies its time quantum (see all about quantums of processes, PAGE 68) into the Sail user table location TIMER. Consider the following procedure, which is roughly equivalent to the one predeclared in Sail:

```
SIMPLE PROCEDURE CLKMOD;
  IF (TIMER←TIMER-1) ≤ 0 THEN INTRPT←-1;
```

To time share several ready processes, one should include polling points in the relevant process procedures and should execute the following statements:

```
INTMAP ( INTCLK_INX, CLKMOD, 0);
ENABLE ( INTCLK_INX );
```

The macro SCHEDULE-ON-CLOCK-INTERRUPTS defined in SYS:PROCES.DEF is equivalent to these two statements. Now, when the time quantum of a process is exceeded by the number of clock ticks since it began to run, the integer INTRPT is set, and this causes the next polling point in the process to **cause a** rescheduling (see about rescheduling and INTRPT on PAGE 70). The current running process will be made ready, and the scheduling algorithm chooses a ready process to run.

16.3 - DEFERRED INTERRUPTS

Deferred Interrupts are processed at the next polling point in your program after the interrupt occurs. Essentially, they are implemented by the provision of a special Immediate interrupt routine that writes some informatton into a special buffer, sets the flag INTRPT, and dismisses itself. (For more details, see the following subsection). Then, when the next polling point is reached, the current process is made ready while a special process (whose procedure is called INTPRO) is run INTPRO will execute any procedures which have been deferred to this point, and then will **call the** scheduler to decide what process is to run next.

One very common use of deferred interrupts is to cause an event soon after some asynchronous condition (say, TTY activation) occurs. This effect **may** be obtained by the following sequence:

```
INTSET(IPRO-NEW,0); COMMENT this will cause
    the interrupt process to be sprouted and
    assigned to IPRO. This process will execute
    procedure INTPRO and will have priority zero
    (the highest possible). ;

INTMAP(<index>,DFRINT,
  DFCPKT(0,<event type>,<event notice>,<cause options>));

ENABLE(<index>);
```

In SYS:PROCES.DEF is the useful macro

```
DEFERRED_CAUSE_ON_INTERRUPT(<index>,
    <event type> , <notice> , <options>)
```

which may be used to replace the INTMAP statement.

16.4 - MORE COMPLICATED DEFERRED INTERRUPTS

This section explains the Runtimes INTSET, INTMAP, and DFRINT in detail and explains how to make more than a simple cause happen at the next polling point following the interrupt.

INTSET
Before any kind of deferred interrupt may be done, an INTSET must be done. It should be done only once per program The statement

```
INTSET( <item_expression>, <options>);
```

sprouts the INTPRO process with the specified <options>. The item of the <item-expression> will become the process item of the INTPRO process. The <options> are the same as those for the SPROUT statement, page 67. However, the default priority for INTPRO is 0, which is the highest possible priority, and is reserved for INTPRO alone. Thus, when rescheduling is done at the first polling point after the Interrupt, INTPRO's high priority will automatically cause it to become the running process.

INTMAP
An INTMAP must be done for each type of interrupt one wants handled (clock, TTY, <esc>I, etc.). To change the way an interrupt is handled, simply do another INTMAP for that type of Interrupt. INTMAP always takes three arguments:

```
INTMAP ( <index> , <simple proc> ,
    <integer expression> )
```

<index> is the code for the type of interrupt (see Interrupt Codes, page 102). <simple proc> is a simple parameterless procedure that will be run at interrupt level whenever an interrupt of type <index> arrives. For deferred interrupts, this will **always be the** predeclared procedure DFRINT. Users who write their own <simple proc>s should observe the restrictions mentioned on page 78. <integer expression> **acts as** a parameter to DFRINT -- more about it later.

INTMAP maintains two tables, both indexed by the interrupt code, <index>. One table is for the <simple proos and the other is for the <integer expression>s. When any enabled interrupt occurs, the Sail interrupt handler sets up some accumulators, then indexes into the table of <simple proc>s, and PUSHJs to the procedure. When the procedure exits, or if no <simple proc> was found, the interrupt handler dismisses itself.

DFRINT
DFRINT sets up a buffer with information **that INTPRO** will use to call the procedure that the user wants run at the next polling point. Such procedures must be specified in a special way.

The user must construct a block of core, called a "calling block", probably by using the MEMORY **and** LOCATION features of Sail, PAGE 25, or Start Code. It must look like:

```
<number of words in the block>
<1st parameter to the procedure>
< second parameter to the procedure>
. , ,
<last parameter to the procedure>
-1 ,,<address of the procedure>
```

For example, one might call FOO(I,J,K) **by saying:**

```
PROCEDURE FOO ( INTEGER i,j,k); . . . ;
. . .
SAFE INTEGER ARRAY FOOBLK [1:5];
. . .
FOOBLK [1] ← 5;
FOOBLK [2] ← I;
FOOBLK [3] ← J;
FOOBLK [4] ← K;
FOOBLK [5] ← (-1 LSH 18)+LOCATION(FOO);
```

NOTE: The procedure specified to INTPRO must not be declared inside any process except the main program. Otherwise, its environment will not be available when INTPRO runs, However, there is a rather complex way to get around this by using <environment>,,PDA as the last word of the calling block. See a Sail hacker if you must do this and don't know what <environment> or PDA mean.

The next step towards specifying FOO to INTPRO is to call INTMAP like so:

```
INTMAP( <index>, DFRINT,
    <AOBJN pointer to calling block> );
```

where <index> is the code for the interrupt that you desire. An AOBJN pointer for a block of core is defined as

```
-<number of words>,,<starting address>
```

Thus to call FOO on a deferred interrupt of, say <esc>I, include the statement

```
INTMAP( INTTTI_INX, DFRINT,
    -5 LSH 18 + LOCATION(FOOBLK[1]);
```

Now, whenever an interrupt of the type specified in INTMAP occurs, DFRINT runs, and uses the table of <integer expression>s to retrieve the AOBJN pointer appropriate for this type of interrupt. Using the AOBJN pointer, DFRINT writes the calling block and some other useful information into a special circular buffer called the Deferred Interrupt buffer. The length of the buffer determines how many interrupts can be queued up waiting to be processed. INTMAP usually initializes the DI buffer to 128 words, which is quite enough unless the program is very slow about processing deferred interrupts (i.e. it doesn't poll very often). A larger DI buffer can be obtained at any time that one is sure the current buffer is empty (i.e. no deferred Interrupts pending) by executing the runtime

```
INTTBL( <size of new DI buffer in words> )
```

DFRINT uses two pointers into the DI buffer: readpt and writept. Whenever it writes a new calling block and etc. into the DI, it begins the writing at the writept and then advances the writept when it's done. When INTPRO reads the DI buffer, it starts at the readpt and continues calling procedures until the writept is reached, updating readpt as it goes. The effect of this is to queue deferred interrupts. Interrupts occuring while INTPRO is active merely add another calling block to the DI which will be processed before the main program is resumed.

When DFRINT is finished writing into the DI buffer, it changes the status of INTPRO from suspended to ready. It sets the INTRPT integer so that the next polling point will cause a rescheduling. The special high priority of INTPRO causes it to be chosen by the scheduler, and it begins to run.

THE DEFERRED INTERRUPT PROCESS - INTPRO
INTPRO first restores the following information which was stored by DFRINT at the time of the interrupt.

LOCATION	CONTENTS
USER	The base of the user table (GOGTAB).
AC 1	Status of spacewar buttons.
AC 2	Your job status word (JBTSTS). See [Moorer-I section II.D.13.
I JBCNI(USER)	XJBCNI(i.e.JOBCNI) at time of interrupt.
IJBTPC(USER)	XJBTPC(i.e.JOBTPC) at time of interrupt.
IRUNNR(USER)	Item number of running process at time of interrupt.

Then INTPRO calles the procedure described by the calling block. When the procedure is finished, INTPRO looks to see if the DI buffer has any more entries left. If it does, INTPRO handles them in the same manner. Otherwise INTPRO suspends itself and the highest priority ready process takes over.

DFRIIN
For those who want more than one procedure to be called as a deferred interrupt for a given interrupt type, the runtime function DFRIIN is provided.

```
DFR1IN ( <AOBJN pointer> );
```

will put another calling block after writept in the DI buffer. This procedure may then be called by an immediate interrupt simple 'procedure. For instance, suppose we want to call FOO and BAZ as defferred Interrupts for <esc>I. This may be done by:

```
SIMPLE PROCEDURE ZORCH;
    BEGIN
    DFR1IN( <AOBJN pointer for FOO call> );
    DFR1IN( <AOBJN pointer for BAZ call> );
    END;
...
INTMAP ( INTTTY_INX, ZORCH, 0 );
ENABLE ( INTTTY_INX );
```

SECTION 17

LEAP AND PROCESS RUNTIMES

We will follow the same conventions for describing Leap execution time routines as were used in describing the runtimes of the Algol section of Sail (see page 3 1).

17.1 - TYPES AND TYPE CONVERSION

TYPEIT

CODE ← TYPEIT (ITM)

The type of the datum linked to an item is called the type of an item. An item without a datum is called untyped. TYPEIT is an integer function which returns an integer CODE for the type of the item expression ITM that is its argument. The codes are:

```
 0 - item deleted or never allocated
     untyped
 2 - Bracketed Triple item
 3 - string
 4 - real
 5 - integer
 6 - set
 7 - list
 8 - procedure item
 9 - process item
10 - event item
11 - context item
12 - reference item
16 - string array
17 - real array
18 - integer array
19 - set array
20 - list array
24 - context array
25 - error (the runtime screwed up)
```

The user is encouraged to use TYPEIT. It requires the execution of only a few machine instructions and can save considerable debugging time.

CVSET

SET ← CVSET (LIST)

CVSET returns a set given a list expression by removing duplicate occurences of items in the list, and reordering the items into the order of their internal Integer representations.

CVLIST

LIST ← CVLIST (SET)

CVLIST returns a list given a set expression. It executes no machine instructions, but merely lets you get around Sail type checking at compile time.

CVN and CVI

INTEGR ← CVN (ITM)
ITM ← CVI (INTEGR)

CVN returns the integer that is the internal representation of the item that is the the value of the item expression ITM. CVI returns the item that is represented by the integer expression INTEGR that is its argument. Legal item numbers are between (inclusively) 1 and 4095, but you'll get in trouble if you CVI when no item has been created with that integer as its representation. Absolutely no error checking is done. CVI is for daring men. See about item implementation, page 54, for more information about the internal representations of items.

17.2 - MAKE AND ERASE BREAKPOINTS

BRKERS, BRKMAK, BRKOFF --

BRKMAK (BREAKPT-PROC)
BRKERS (BREAKPT-PROC)
BRKOFF

In order to give the programmer some idea of what is going on in the associative store, there is a provision to interrupt each MAKE and ERASE operation, and enter a breakpoint procedure. The user can then do whatever he wants with the three items of the association being created or destroyed. ERASE Foo ⊗ ANY ≡ ANY will cause the breakpoint procedure to be activated once for each association that matches the pattern. MAKE it1 ⊗ it2 ≡[it3 ⊗ it4 ≡ it5] will cause the breakpoint procedure to be activated twice.

The user's breakpoint procedures must have the form:

```
PROCEDURE Breakpt_proc ( ITEMVAR a, o, v )
```

If the association being made or erased is A⊗O≡V, then directly before doing the Make or Erase, Breakptgroc is called with the items A, 0, and V for the formals a, o, and v.

To make the procedure Breakpt_proc into a breakpoint procedure for MAKE, call BRKMAK with Breakptgroc as a parameter. To make the procedure Breakpt_proc rnto a breakpoint procedure for ERASE, call BRKERS with Breakpt_proc as its parameter. To turn off both breakpoint procedures, call BRKOFF with no parameters

NOTE: BRKMAK, BRKERS and BRKOFF are not predeclared. The user must include the declarations:

```
EXTERNAL PROCEDURE BRKERS ( PROCEDURE BP );
EXTERNAL PROCEDURE BRKMAK ( PROCEDURE BP );
EXTERNAL PROCEDURE BRKOFF;
```

17 3 - PNAME RUNTIMES

CVIS

"PNAME" ← CVIS (ITEM , @FLAG);

The print name of ITEM is returned as a string. Items have print names only if one includes a REQUIRE n PNAMES statement in his program, where n is an estimate of the number of pnames the program will use An Item's print name is the identifier used to declare it, or that pname explicitly given it by the NEW-PNAME function (see below). FLAG is set to False (Ø) if the appropriate string is found. Otherwise it is set to TRUE' (-1), and one-should not put great faith in the string result.

CVSI

ITEM ← CVSI("PNAME", @FLAG);

The Item whose pname is the same as the string argument PNAME is returned and FLAG is set to FALSE if such an ITEM exists. Otherwise, something very random is returned, and FLAG is set to TRUE.

DEL-PNAME

DEL-PNAME (ITEM)

This function deletes any string PNAME associates with this ITEM.

NEW-PNAME

NEW-PNAME (ITEM , "STRING");

This function assigns to the Item the name "STRING". Don't perform this twice for the same Item without first deleting the previous one. The corresponding name or Item may be retrieved using CVIS or CVSI (see above). The NULL string is prohibited as the second argument.

17 4 - OTHER USEFUL RUNTIMES

LISTX

VALUE ← LISTX (LIST , ITEM , N)

The value of this integer function is 0 if the ITEM (an item expression) does not occur in the list at least N (an integer expression) different times in the LIST (a list expression). Otherwise LISTX is the index of the Nth occurrence of ITEM in LIST. For example,

```
LISTX ({{Foo, Baz, Garp, Baz}}, Baz, 2) is 4.
```

FIRST, SECOND, THIRD

ITEM ← FIRST (BRAC-TRIP-ITEM)
ITEM ← SECOND (BRAC-TRIP-ITEM)
ITEM ← THIRD (BRAC-TRIP-ITEM)

The Item which is the FIRST, SECOND, or THIRD element of the association connected to a bracketed triple item (BRAC-TRIP-ITEM) is returned. If the item expresslon BRAC_TRIP_ITEM does not evaluate to a bracketed triple, an error messages issues forth.

LOP

ITEM ← LOP (SETVARIABLE);
ITEM ← LOP (LISTVARIABLE);

LOP will remove the first item of a set or list from the set or list, and return that item as its value. Note that the argument must be a variable because the contents of the set or list is changed. If one LOPs an empty set or a null list, an error message will be issued.

COP

ITEM ← COP (SETEXPR);
ITEM ← COP (LISTEXPR);

COP will return the first item of the set or list just as LOP (above) will. However, it will NOT remove that item from the set or list. Since the set or list will be unchanged, COP's argument may be a set or list expression. As with LOP, an error message will be returned if one COPs an empty set or a null list.

LENGTH

VALUE ← LENGTH (SETEXPR);
VALUE ← LENGTH (LISTEXPR);

LENGTH will return the number of items in that set or list that is its argument, LENGTH(S) = 0 is a much faster test for the null set or list that S = PHI or S = NIL.

SAMEIV

VALUE ← SAMEIV (ITMVAR1 , ITMVAR2);

SAMEIV is useful in Matching Procedures to solve a particular problem that arises when a Matching Procedure has at least two ? itemvar arguments. An example will demonstrate the problem:

```
FOREACH X | Matchingproc( X, X ) DO . . . ;
FOREACH X, Y | Matchingproc( X, Y ) DO . . . ;
```

Clearly, the matching procedure with both arguments the same may want to do something different from the matching procedure with two different Foreach itemvars as its arguments. However, there is no way inside the body of the matching procedure to differentiate the two cases since in both cases both itemvar formals have the value BINDIT. SAMEIV will return True only in the first case, namely 1) both of its arguments are ? itemvar formals to a matching procedure, 2) both had the same Foreach itemvar passed by reference to them. It will return False under all other conditions, including the case where the Foreach itemvar is bound at the time of the call (so it is not passed by reference, but its item value is passed by value to both formals).

17.5 - GENERAL PROCESS RUNTIMES

MY PROC

PROCITEM ← MYPROC

MYPROC returns the process item of the process that it is executed in. If it is executed not inside a process, then MAINPI (the item for the main process) is returned.

CALLER

PROCITEM ← CALLER (PROCITEM2)

CALLER returns the process item of the process that most recently resumed the process referred to PROCITEM2. PROCITEM2 must be the process item of an unterminated process, otherwise an error message will be issued. If PROCITEM2's process has never been called, then the process item of the process that sprouted PROCITEM2 is returned.

MKEVTT

MKEVTT (ITEM)

MKEVTT will convert its item argument to an event type item. The old datum will be overwritten. The type of the item will now be "event type". Any item except an event type item may be converted to an event type item by MKEVTT.

PRISET

PRISET (PROCITM , PRIORITY)

PRISET sets the priority of the process specified by PROCITM (an item expression that must evaluate to the process item of a non-terminated process) to the priority specified by the integer expression PRIORITY. Meaningful priorities are the integer between 1, the highest priority, to 15, the lowest priority. Whenever a rescheduling is called for, the scheduler finds the highest priority class that has at least one ready process in it, and makes the first process on that list the running process. See about the scheduler, page 70.

PSTATUS

PRIORITY ← PSTATUS (PROCITM)

PSTATUS returns an integer indicating the status of t tie process specified by the item expression PROCI TM

-1	running
0	suspended
1	ready
2	terminated

URSCHD

URSCHD

URSCHD is essentially the Sail Scheduler When one calls URSCHD, the scheduler finds the highest priority class that has at leas+ one Ready process in it. Each class has a list of processes associated with it, and the scheduler choses the firs? ready process on the list. This process then becomes the running process and is put on the end of the list. If no processes have ready status, the scheduler looks to see if the program is enabled for any interrupts. If the program is enabled for some kind of interrupt that may still happen (not arithmetic overflow, for Instance), then the scheduler puts the program into interrupt wait. After the intrrupt is dismissed, the scheduler tries again to find a ready process. If no Interrupts that may still happen are enabled, and there are no ready processes, the error message "No one to run" is issued.

17.6 - RUNTIMES FOR USER CAUSE AND INTERROGATE PROCEDURES

SETCP AND SETIP

SETCP(ETYPE , PROC-NAME)
SETCP (ETYPE , DATUM (PROC-ITEM))
SETIP (ETYPE , PROC-NAME)
SETIP (ETYPE , DATUM (PROC_ITEM))

SETCP and SETIP associate with the event type specified by the item expression ETYPE a procedure specified by its name or the datum of a procedure item expression.

After the SETCP, whenever a Cause statement of the specified event type is executed, the procedure specified by PROC-NAME or PROC-ITEM is called. The procedure must have three formal parameters corresponding to the event type, event notice, and options words of the CAUSE statement. For example,

```
PROCEDURE CAUSEIT ( ITEMVAR ETYP, ENOT;
          INTEGER OP );
```

After SETIP, whenever an Interrogate statement of the specified event type is executed, the procedure specified by PROC-NAME or PROC-ITEM is called. The procedure must have two formal parameters corresponding to the event type and options words of the Interrogate statement and return an item. For example,

```
ITEM PROCEDURE ASK_IT ( ITEMVAR ETYP;
          INTEGER OP );
```

It is an error if a Cause or Interrogate statement tries to call a procedure whose environment (static - as determined by position of its declaration, and dynamic - as detet mined by the execution of the SETCP or SETIP) has been exited.

See page 74 and page 74 for more information on the use of SETCP and SETIP, respectively.

CAUSE 1

ITMVAR ← CAUSE1 (ETYPE , ENOT , OPTIONS)
ITMVAR ← CAUSE1 (ETYPE , ENOT)
ITMVAR ← CAUSE 1 (ETYPE)

CAUSE1 is essentially the procedure executed for CAUSE statements if no SETCP has been done for the event type ETYPE. See the description of the Sail defined Cause statement, page 74, for further elucidation.

ASKNTC

ITMVR ← ASKNTC (ETYPE , OPTIONS)
ITMVR ← ASKNTC (ETYPE)

ASKNTC is the procedure executed for INTERROGATE statements if no SETIP has been done for the event type ETYPE. See the description of the Sail defined Interrogate statement, page 75, for further elucidation.

ANSWER

BITS ← ANSWER (ETYPE , ENOT , PROC-ITEM)

ANSWER will attempt to wake up from an interrogate wait the process specified by the item expression PROC_ITEM. If the process is not in a suspended state, Answer will return an integer with the bit '400000 in the right half (NOJOY in SYS:PROCES.DEF) turned on. If the process is suspended, it will be made ready, and removed from any wait queues it may be on. The bits corresponding to the options word of the interrogate statement that put it in a wait state will be returned. Furthermore, if the SAY-WHICH bit was on, the appropriate association, namely EVENT-TYPE ⊗ ENOT ≡ ETYPE, will be made. See page 74 for more information on the use of ANSWER.

SECTION 18

BASIC CONSTRUCTS

18.1 - SYNTAX

```
<variable>
    ::= <identifier>
    ::= <identifier> [ <subscript-list> ]
    ::= DATUM ( <typed-item-expression> )
    ::= DATUM ( <typed-item-expression> )[
        <subscript-list> ]
    ::= PROPS ( <item-expression> )
    ::= <context-element>


<typed_item_expression>
    ::= <typed_itemvar>
    ::= <typed-item>
    ::= <typed_itemvar_procedure>
    ::= <typed_item_procedure>
    ::= <typed_itemvar_array>
        [ <subscript-list> ]
    ::= <typed_item_array>
        [ <subscript-list> ]
    ::= <itemvar> ← <typed-item-expression>
    ::= IF <boolean-expression> THEN
        <typed-item-expression> ELSE
        <typed-item-expression>
    ::= CASE <algebraic_expression> OF (
        <typed-item-expression-list> )


<typed-item-expression-list>
    ::= <typed-item-expression>
    ::= <typed_item_expresssion_list> ,
        <type_item_expression>

<subscript_list>
    ::= <algebraic_expression>
    ::= <subscript_list> ,
        <algebraic-expression>
```

18.2 - SEMANTICS

VARIABLES
If a variable is simply an identifier, it represents a single value of the type given in Its declaration.

If it is an identifier qualified by a subscript list it represents an element from the array bearing the name of the Identifier However, an identifier qualified by a subscript list containing only a single subscript may be either an element from a one dimensional array, or an element of a list. Note that the token "∞" may be used in the subscript expression of a list to stand for the length of the list, e.g. LISTVAR[∞-2]←LISTVAR[∞-1]

The array should contain as many dimensions as there are elements in the subscript list. A[I] represents tne I+1th element of the vector A (if the vector has a lower bound of Ø). B[I,J] is the element from the Itlth row and Jt 1 th column of the two-dimensional array B. To explain the indexing scheme precisely, all arrays behave as if each dimension had Its origin at 0, with (integral) indices extending infinitely far in either direction However, only the part of an array between (and Including) the lower and upper bounds given in the declaration are available for use (and in fact, these are the only parts allocated). If the array is not declared SAFE, each subscript is tested against the bounds for Its dimension If it is outside its range, a fatal message is printed identifying the array and subscript position at fault SAFE arrays are not bounds-checked. Users must take the consequences of the journeys of errant subscripts for SAFE arrays. The bounds checking causes at least three extra machine instructions (two of which are always executed for valid subscripts) to be added for each subscript in each array reference. The algebraic expressions for lower and upper bounds in array declarations, and for subscripts in subscripted varrablcs, are always converted to Integer values (see page 2 1) before use.

For more information about the implementation of SAIL arrays, see page 106.

DATUMS
DATUM(X) where X is a typed item expression, will act exactly like a variable with the type of the item expression. The programer is responsible for seeing that the type of the item is that which the DATUM construct thinks it is For example, the Datum of a Real Itemvar will always Interpret the contents ot the Datum location as a floating point number even if the program has assigned a string item to the Real I temvar.

PROPS
The PROPS of an item will always act as an integer variable Any algebraic value assigned to a props will

be coerced to an integer (see about type conversions, page 2 1) then the low order 12 bits will be stored in the props of the item. Thus, the value returned from a props will always be a non-negative integer less than '7777 (4095 in decimal).

IDENTIFIERS

You will notice that no syntax was included for the non-terminal symbols <identifier> or <constant>. It is far easier to explain these constructs in an informal manner.

A SAIL letter is any of the upper or lower case letters A through Z, or the underline character (_ or !), they are treated equivalently). Lower case letters are mapped into the corresponding upper case letters for purposes of symbol table comparisons (SCHLUFF is the same symbol as Schluff). A digit is **any of the** characters 0 through 9.

An identifier is a string of characters consisting of a letter followed by virtually any number of ietters **and** digits There must be a character which is neither a letter nor **a** digit (nor either of the characters "." or "$") both before and after every identifier. In other words, if YOU can't determine where one identifier ends **and** another begins in a program you have never seen before, well, neither can SAIL.

There is a set of identifiers which are used as SAIL delimiters (in the Algol sense -- that is, BEGIN is treated by Algol as if it were a single character. Such an approach is not practical, so a reserved identifier is used). These identifiers are called Reserved Words and may not be used for any purpose other than those given explicitly in the syntax, or in declarations (DEFINES) which mask their reserved-word status over the scope of the declarations. E.g., "INTEGER BEGIN" is allowed, but a Synonym (see page 9) should **have been** provided for BEGIN if any new blocks are desired within this one, because BEGIN is ONLY an Integer in this block. Another set of identifiers have preset declarations -- these are the execution time functions. These latter Identifiers may also be redefined by the user: they behave as if they were declared in a block surrounding the outer block. A list of reserved words may be found in Appendix 2. A list of predeclared identifiers may be found in the Appendix 3. It should be noted that due to the stupidity of the parser, it is impossible to declare certain reserved words to be identifiers. For example, INTEGER REAL; will give one the syntax error "Bogus token in declaration".

Some of the reserved words are equivalent to certain special characters (e.g. "|" for "SUCH THAT"). A table of these equivalences may be found in Appendix 4.

ARITHMETIC CONSTANTS

12369	Integer with decimal value 12369
'12357	Integer with octal value 12357
123.	Real with floating point value 123.0
0123.0	Real with floating point value 123.0
.524	Real with floating point value 0.524
5.3@2	Real with floating point value 530.0
5.342@-3	Real with floating point value 0.005342

The character ' (right quote) precedes a string of digits to be converted into an OCTAL number.

If a . or a @ appears in a numeric constant, the type of the constant is returned as Real (even if it has an integral value). Otherwise it is an integer. Type conversions are made at compile time to make the type of a constant commensurate with that required by a given operation. Expressions involving only constants are evaluated by the compiler and the resultant values are substituted for the expressions.

The reserved word TRUE is equivalent to the Integer (Boolean) constant -1; FALSE is equivalent to the constant 0.

STRING CONSTANTS

A String constant is a string of ASCII characters (any which you can get into a text file) delimited at each end by the character ". If the " character is desired in the string, insert two " characters (after the initial delimiting " character, of course).

A String constant behaves like any other (algebraic) primary. It is originally of type String, but may be converted to Integer by extracting the first character if necessary (see page 21).

The reserved word NULL represents a String constant containing no characters (length=0).

Examples: The left hand column in **the** table that follows gives the required input

INPUT	RESULT	LENGTH
"A STRING"	A STRING	8
"WHAT'S ""DOK"" MEAN?"	WHAT'S "DOK" MEAN?	18
"""A QUOTED STRING"""	"A QUOTED STRING"	17
""		0
NULL		0

COMMENTS

If the scanner detects the identifier COMMENT, all characters up to and Including the next semicolon (;) will be ignored. A comment may appear anywhere as long as the word COMMENT is properly delimited (not in a String constant, of course);

A string constant appearing just before **a** statement also has the effect of a comment.

SECTION 19

USING SAIL

19.1 - FOR BEGINNERS

If you simply want your Sail program compiled, loaded, and executed, do the following:

1. Create a file with your program on it named "XXXXXX.SAI" where "XXXXXX" may be any name you wish.

2. Get your job to monitor level, and type "EXECUTE XXXXXX".

3. The RPG system will type back at you "SAIL: XXXXXX", and start Sail. When Sail hits a page boundry in your file, it will type "1" or whatever the number of the page that it is starting to read.

4. When the compilation is complete, Sail will type "LOADING".

5. When the loading is complete, the loader will type "LOADER nK CORE" where n is your core size. Sail will then type "EXECUTION".

6 When execution is complete, Sail will type "END OF SAIL EXECUTION" and exit.

At any time during 3 through 6 above, you could get an error message from the system such as "ILL MEM REF", "ILLEGAL UUO" etc. followed by some core locations. These are Sail bugs. You will have to see a Sail hacker about them, or attempt to avoid them by rewriting the offense part of your program, or try again tomorrow.

If you misspell your file, RPG will complain "UNKNOWN FILE: YYYYYY" where "YYYYYY" is your misspelling. Otherwise, the error messages you receive during 3 above will be compilation errors (bad syntax, type mismatch, begin-end mismatch, unknown identifiers, etc). See Section 19 about these.

If you get through compilation (step 3) with no error messages, the loading of your program will rarely fail. If it somehow does, it will tell you. See a Sail hacker about these

If you also get through loading (step 4) with no errors, you aren't yet safe Sail will give you error messages dur ing the execution of your program if you exceed the bounds of an array, exceed string space, etc. See Section 19 about these too.

If you never get an error message, and yet you don't get the results you thought you'd get, then you've probably made some mistakes in your programing. Use RAID or DDT and Section 19 to follow your program as it executes, and see where it goes wrong (or else guess at it) It is quite rare for Sail to have compiled runable but incorrect code from a correct program. The only way to ascertain whether this is the case is to Isolate the section of your program that is causing Sail to generate the bad code, and then patiently step through it instruction by instruction using RAID or DDT, and check to see that everything it does makes sense.

19.2 - THE COMPLETE USE OF SAIL

The general sequence of events in using Sail is:

1. Start Sail

2. Compile one or more files into one or more binary files, with possibly a listing file generated.

3. Load the binary file(s) with the appropriate upper segment or with the Sail runtime library, and possibly with RAID or DDT.

4. Start the program, possibly under the control of RAID or DDT.

5. Let the program finish, or stop it to examine the core with RAID or DDT, or to reallocate storage with the REENTER command.

Starting Sail is automatic with the RPG commands described below. Otherwise, "R SAIL" will do.

19.3 - COMPILING SAIL PROGRAMS

If one started Sail with "R SAIL", then Sail will type back an "*" at you and wait for you to type in a <command line> It will do the compilation specified by that command line, then ask for another, and so on until you type "LOADER!" Instead of a command line. At this point it will call the Loader.

If you use RPG, follow the RPG command with a list of <command line>s separated by commas. The compilation of each <command line> will be done before the next <command line> is read and processed. The RPG commands are:

EXecute	compile, load, start
TRY	compile, load with RAID or DDT, start
DEBug	compile, load with RAID or DDT, start RAID or DDT
LOAd	compile, load
PREPare	compile, load with RAID or DDT
COMpile	compile

See [Moorer] for more information about the use of RPG and the switches available to it.

COMMAND LINE SYNTAX

```
<command-line>
    ::= <binary-name> <listing-name> ←
        <source-list>
    ::= <file_spec>@
    ::= <file_spec> EXC

<binary-name>
    ::= <file_spec>
    ::= <empty>

<listing_name>
    ::= , <file_spec>
    ::= <empty>

<source-list>
    ::= <file_spec>
    ::= <source-list> , <file_spec>

<file_spec>
    ::= <file-name> <file_ext> <proj_prog>
    ::= <device-name> <file_spec> <switches>
    ::= <device-name> <switches>

<file-name>
    ::= <legal_sixbit_id>

<file_ext>
    ::= <legal_sixbit_id>
    ::= <empty>

<proj_prog>
    ::= [ <legal_sixbit_id> ,
          <legal_sixbit_id> ]
    ::= <empty>

<device_name>
    ::= <legal_sixbit_id>

<switches>
    ::= ( <unslashed_switch_list> )
    ::= <slashed-switch-list>
    ::= <empty>

<unslashed_switch_list>
    ::= <switch_spec>
    ::= <unslashed_switch_list> <switch_spec>

<slashed-switch-list>
    ::= / <switch_spec>
    ::= <slashed-switch-list> / <switch_spec>

<switch_spec>
    ::= <valid_switch_name>
    ::= <signed-integer> <valid-switch-name>

<valid-switch-name>
    ::= D
    ::= L
    ::= M
    ::= P
    ::= Q
    ::= R
    ::= S
    ::= C
    ::= F
    ::= K
```

COMMAND LINE SEMANTICS

All this is by way of saying that SAIL accepts commands in essentially the same format accepted by DEC processors such as MACRO and FORTRAN. The binary file name is the name of the output device and file on which the ready to load object program will be written. The listing file, if included, will contain a copy of the source files with a header at the top of each page and an octal program counter entry at the head of each line (see page 92). The listing file name is often omitted (no listing created). The source file list specifies a set of user-prepared files which, when concatenated, form a valid SAIL program (one outer block).

legal-sixbit-identifier is a name which is acceptable to the time sharing system as a valid file name, device name, extension, etc. when its first six (device, file) or three (extension, project-programmer number) are converted from ASCII to SIXBIT. For more information about file and device names, see [Moorer].

If file_ext is omitted from the binary-name, the extension for the output file will be REL. The default extension for the listing file is .LST SAIL will first try to find source files under the names given. If this fails, and the extension is omitted, the same file with a .SAI extension will be tried.

If device-name is omitted, DSK. is assumed. If proj_prog is omitted, the project-programmer number for the job is assumed.

Switches are parameters which affect the operation of the compiler A list of switches may appear after any file name The parameters specified are changed immediately after the file name associated with them is processed. The meanings of the switches are given below.

The binary, listing and (first) source file names are processed before compilation -- subsequent source names (and their switches) are processed whenever an end-of-file condition is detected in the current source file. Source files which appear after the one containing the outer block's END delimiter are not Ignored, but should contain only comments.

Each new line in the command file (or entered from the teletype) specifies a separate program compilation. Any number of programs can be compiled by the same SAIL core image.

The f ile_spec@ command causes the compiler to open the specified file as the command file. Subsequent commands will come from this file. If any of these commands is file_spec@, another switch will occur.

The file-spec EXC command will cause the specified file to be run as the next processor. This program will be started in "RPG mode". That is, it will look on the disk for its commmands if its standard command file is there -- otherwise, command control will revert to the TTY. The default option for this file name is .DMP. The default device is SYS.

SWITCHES

The following table describes the SAIL parameter switches. If the switch letter is preceded in the table by the D character, a decimal number is expected as an argument. 0 is the default value. The character 0 indicates that an octal number is expected for this switch. Otherwise the argument is ignored.

ARG SWITCH FUNCTION

C — This switch turns on CREFfing. The listing file (which must exist) will be in a format suitable for processing by CREF, the program which will generate a cross-reference listing of your SAIL program from your listing files.

D — For every occurrence of this switch in the command line, the amount of space for the push down stack used in expanding macros (see page 46) is doubled. Use this switch if the compiler indicates to you that this stack has overflowed. This shouldn't happen unless you nest DEFINE calls extremely deeply.

0 F — 0 is an octal number which specifies exactly what kind of listing format is generated. 0 contains information about 5 separate listing features, each of which is assigned a bit in 0.

```
1 List the program counter
  (see / L switch below).
2 List with line numbers from
  the source text.
4 List the macro names before
  expansion.
10 Expand macro texts in the
  listing file.
20 Surround each listed macro
  expansion with ⊂ and ⊃ .
```

The compiler is initialized with /6f (i.e. list line numbers and macro names).

H — This switch is used to make your program sharable. When loaded, the code and constants will be aced in the second (write-protected) segment, while data areas will be allocated in the lower, non-shared segment. Load such programs like this: Run the loader directly, then respond: *{ddt switches} p r o g n a m e {,other prognames} /LSYS:HLBSAn/G<crlf> The sharable library HLBSAn is identical to LIBSAn, except that it expects to run mostly in the upper (shared) segment. Recall that n is the current version number. When you have finished loading, in order to write-protect the sharable portion, you'll have to deposit (by hand) the following instructions:

LOCATION	INSTRUCTION	EXPLANATION
134	211000 1	(MOVNI 0,1)
135	47000 36	(CALLI 36)
136	254200 0	(HALT)
137	47000 12	(CALLI 12)

Then type: START 134, and SSAVE it when it exits (worry if it HALTS). This feature

should be used only if you have **a program** which is likely to be used by **a lot of** people at once.

O L In compiling a SAIL program, an internal variable called PCNT (for program counter) is incremented (by one) for each word of-- code generated. This value, initially 0, represents the address of a word of code in the running program, relative to the load point for this program. The current **octal** value of PCNT plus the value of another internal variable called LSTOFFSET, is printed at the beginning of each output line in a listing file. For the first **program** compiled by a given SAIL core image, LSTOFFSET is initially 0. If the L switch occurs in the command and the **value O** is non-negative, **O replaces the current value of** LSTOFFSET. If O is -1, the current size of DDT is put into LSTOFFSET. If O is -2, the current size of RAID is used. In "RPG mode" the final value of PCNT is added to LSTOFFSET after each compilation. Thus by deleting all .REL files produced by SAIL, and by compiling all SAIL programs which are to be loaded together with one RPG command which includes the L switch, you can obtain listing files such that **each** of these octal numbers represents the actual starting core address of the code produced by the line it precedes. At the time of this writing, RPG would not **accept** minus signs in switches to be sent to **processors. Keep trying.**

P Each occurrence of this switch doubles the size of the system push down list. It has never been known to overflow.

Q Each occurrence doubles the size of the String push down list. No trouble has been encountered here, either.

R Each occurrence doubles the size of the compiler's parsing and semantic stacks. A long conditional statement of the form (IF ... THEN ... ELSE IF ... THEN ... ELSE IF ...) has been known **to cause these stacks to** overflow their normally allocated sizes.

D S The size of String space is Set to D words. String space usage is a function of the number of identifiers, especially macros, declared by the user. In the rare case of String space exhaustion, 5000 is a good first number to try.

K The counter mechanism of Sail is activated, enabling one to determine the frequency of execution of each statement in **your Sail** program. See appendix 12, the Statement Counter System. This switch is ignored unless a listing is specified with **a** /LIST.

Here is an example of a compile string which a user who just has to try every bell and whistle **available** to him might type to compile **a** file named NULL:

```
COMPILE /LIST /SAIL NULL(RR-2L5000S)
```

The switch information contained in **parentheses** will be sent unchanged to SAIL. Note the convention which allows one **set of parentheses** enclosing a myriad of switches to **replace a "/** character inserted before each one. This string tells the compiler to compile NULL using parse and semantic stacks four times larger than **usual (RR).** A listing file is to be made which assumes that RAID will be loaded and NULL will be loaded right **after** RAID (-2L). His program is big enough to need 5000 words of String space (5000S).

19.4 - LOADING SAIL PROGRAMS

Load the main program, **any separately compiled** procedure files (see page 10), **any assembly language** (see page 11) or Fortran procedures, and DDT or RAID if desired. This is all automatic if you use the LOAD or DEBUG or EXECUTE system commands (see [Moorer]). Any of the SAIL execution time routines requested by your program will be searched **out and loaded** automatically from SYS:LIBSAn.REL. If the shared segment (SYS:SAISEG, etc.) is available **and desired,** type SYS:SAILOW as as your very first LOADER command (before /D even). **Stanford people can** abbreviate SYS:SAILOW as /Y. All this is done automatically by RPG at Stanford.

19.5 - STARTING SAIL PROGRAMS

For most applications, SAIL programs can by started using the START, RUN, EXECUTE, or TRY system commands, or by using the $G command of DDT (RAID). The SAIL storage areas will be initialized. This means that all knowledge of I/O activity, **associative data** structures, strings, etc. from any previous **activation of** the program will be lost. All strings (except constants) will be cleared to NULL. All compiled-in arrays will not be reinitialized (PRELOADed arrays **are preloaded** at compile time - OWN arrays are never initialized). Then execution will begin with the first statement in the outer block of your main program. As each block is entered, its arrays will be cleared as they are allocated. Variables are not cleared. The program will exit when it leaves this outer block.

STARTING THE PROGRAM IN "RPG" MODE

SAIL programs may be started at one of two consecutive locations: at the address contained in the cell JOBSA in the job data area, or at the address just following that one. The global variable RPGSW is set to 0 in the former case, -1 in the latter. Aside from this, there is no difference between the two methods, This cell may be examined by declaring **RPGSW as an** EXTERNAL INTEGER.

19.6 - STORAGE REALLOCATION WITH THE REENTER COMMAND

The compiler dynamically allocates working storage for its push down lists, symbol tables string spaces. etc. It normally runs with a standard allocation adequate for most programs. Switch settings given above **may be** used to change these allocations If desired, these allocations may also be changed by typing TC, followed by REE (reenter). The compiler will ask you if you want to allocate. Type Y to allocate, N to use the standard allocation, and any other character to use **the standard** allocations and print out what they are. All entries will be prompted. Numbers should be decimal. Typing **alt-**mode instead of CR will cause standard allocation to **be** used for the remaining values. The compiler will then start, awaiting command input from the teletype.

For Stanford "Global Model" users, the REE **command** will also delete any REQUIREd or previously typed segment name information. The initialization sequence will then ask for new names.

SECTION 20

DEBUGGING SAIL PROGRAMS

20.1 - ERROR MESSAGES

If the compiler detects **a syntax or semantic error** while compiling a program it will provide the user with the following information:

1) The error message. These **are** English phrases or sentences which attempt to **diagnose the problem. If a message is vague** it is because no specific test for the error has been made and a catchall routine detected it If the message begins with the word "DRYROT" it means that there is **a bug** in the compiler which some strangeness in your program was able to tickle. See a system programmer about this.

2) The current input line. Page and line number, along with the text of the line being scanned, are typed. If the console device is **a TTY, a** line feed will occur at the point in the line just following the last program element scanned. If the device is a DPY, the line will be displayed with a vertical arrow below the scan position. **The absence of a position** indicator means that a macro (DEFINE) body is being expanded.

3) A question mark or arrow (→ or ↑).

Respond to the prompt in any of the following ways:

<cr> Try to continue compilation. A message will be printed and the sequence reentered if recovery is impossible (if a "?" was typed instead of an arrow).

C same as <cr>

<lf> Try to continue the compilation, but don't stop for user response after future errors. I.e. automatic continuation. Messages will fly by (at an unreadable rate on DPYs) until the compilation is complete or an error occurs from which no recovery is possible. In the latter case the question sequence is reentered.

A same as <lf>

S Restart. Sometimes useful if you **are** debugging the compiler (or if you were compiling the wrong file). **The program is** restarted, accepting compilation **commands** from the TTY.

X Exit. All files are closed in their current state. The program exits to the system.

E Edit. This command must be followed by a carriage return, or a space, a filename (in standard format, assumes DSK) and a carriage return, If the filename is missing, the SOS editor (see [Savitzky]) is **started,** given instructions to edit the current source file and to move the editing pointer to the current page and line number. If a file name is present, that file is edited **starting at the** beginning. This feature is available outside Stanford only if the SOS editor is available, and is modified to read a standard CCL file for its input.

T TV edit, Same as E only the TV editor is used (Stanford only).

D Enter DDT or **RAID if one is loaded.** Otherwise, type "NO DDT **LOADED" and re-**quest ion.

Any other character will cause the error routines to spew forth a summary of this table **and re-enter the** quest ion sequence.

ERROR MODES
The above procedure can be modified slightly by setting various modes. One sets a mode by including the appropriate letter before the response. Any of the four modes may be reset by including a minus sign (-) before them. E.g. "-Q". Error modes can **also be set** with the construct REQUIRE **<string_const>** ERROR-MODES. When the compiler sees this in **any** of the source files one is compiling, it reads through the string constant and sets the modes as it sees their letters. These modes remain in effect until the end of the compilation or until reset with a response to **an** error message, or another require error-modes.

The available modes are:

K KEEP type-ahead. Normally, the error handler will flush the input buffer before looking for response characters. This mode allows one to type ahead.

Q QUIET. If the error is continuable, none of the above will be typed. However, you will always be notified of a non-continuable error.

L LOGGING. The first and second items of the

error message will be sent to a file named <prognam>.LOG where <prognam> is the name of the tile of the main program. If you would rather have another name, use F<file specification>, where <file specification> must be a legal file name and PPN. The default extension is .LOG and the default PPN is that of the job. The .LOG file (or whatever it's called) is closed when one's program finishes compilation, or the compilation is terrninated with the S, X, E, or T responses.

N NUMBERS. This mode causes the message 'CALLED FROM xxxx LAST SAIL CALL AT yyyy" to be typed before tne question mark or arrow. Useful to compiler debuggers and hand coders.

Note that setting a mode does nothing by set a mode; it does not cause continuation.

STOPPING RUNAWAY COMPILATIONS
Typing <esc>I will immediately cause the Q and A modes to be reset so that the next error will (a) be typed, and (b) wait for a response rather than continuing automatically.

EXECUTION TIME ERROR MESSAGES
Error messages have nearly the same format as those from the compiler (page 95). They indicate that

1) an array subscript has overflowed;

2) a case index is out of range;

3) a stack has overflowed while allocating space for a recursive procedure; or

4) one of the execution time routines has detected an error.

In Numbers mode, the "CALLED FROM" address identifies, in the first 3 cases, the location in the user program where the error occurred ; the "LAST SAIL CALL AT" address gives the location of the faulty call on the SAIL routine for type 4 messages.

All the replies to error messages described in page 95 are valid. If no file name is typed with the "E" or "T" option, the editor re-opens the last file mentioned in the EDIT system command.

The function USERERR may be used to activate the SAIL error message mechanism. Facilities are provided for changing the mode. See page 42 for details.

20.2 - DEBUGGING

The code output for SAIL programs is designed to be fairly easy to understand when examined using the DDT debugging language or Stanford's display oriented RAID program. A knowledge of the debugger you have chosen is required before this section will be comprehensible.

SYMBOLS
Only those symbols which have been declared INTERNAL (see page 10) and those declared in the currently open "program" are available at a given time. The name of a SAIL program as far as DDT or RAID (henceforth DDRAID) is concerned is the name of the outer block of that program. If no name is given for this block, the name M will be the default

Only the first six non-blank characters of a block name or Identifier will be used in forming a DDRAID symbol. If two identifiers in the same block have the same first six characters the program using them will not get confused, but the user might when trying to locate these identifiers.

To obtain symbols for the execution time routines, load RUNTIM.REL (available from your friendly local SAIL maintainer) with your other files. The routines wilt be loaded from this file, which includes symbols, instead of from the LIBSAI library or shared segment, which do not. Your program will be several thousand words longer when this file is used.

BLOCKS
All block names and identifiers used as variables, procedures or labels in a given (main or separate procedure) program are available for typeout when that program is "open" (NAME$: has been typed). To refer to a symbol, type BLOCK-NAME&SYMBOL/ (substitute ; for / in RAID). The block name may be omitted if you have "opened" the block with BLOCK-NAME$&. The symbol table is block-structured only to the extent that block names have appeared in the source program. For instance, in the program

```
BEGIN "NAME1"
  INTEGER I,J;
  ...
  BEGIN
    INTEGER I,K;
    ...
  END;
  ...
END "NAME1"
```

the symbols J. K, and both symbols I are considered by DDRAID to belong in the same block. Therefore confusion can result with respect to I. This approach was taken to avoid the necessity of generating meaningless block names for DDRAID when none were given in the source program. A compound statement

will be considered by DDRAID to be a block if it has a name.

SAIL GENERATED SYMBOLS
Some extra symbols are generated by SAIL and will show up when you are using DDRAID. They are:

ACS The accumulators P (system push down list pointer), SP(string push down pointer), and TEMP (commonly used temporary) are given symbolic names. Currently, P=' 17, SP=' 16, TEMP=' 14.

OPS The op codes for the UUOs ERR., ERROR., FIX, FLOAT, PDLOV, and ARERR (subscript overflow UUO) are included to make these easy to detect in the code.

ARRAYS For each array declared in the outer block (built-in arrays), the fixed address of its first element is given a symbolic name. This name is constructed from the characters of the array name (up to the first 5) followed by a period. For instance, the first element of array CHT is CHT.; the first element of PDQARR is PDQAR.; The last semicolon was really a period. This dotted symbol points to the second word of the first descriptor for String Arrays (see page 107, page 106).

STRINGS For each string declared in the outer block (built-in strings), the second word of the two word string descriptor is given the name of the string variable, truncated to six letters. The first word of the string descriptor is given a name consisting of the first five letters of the string's name followed by a period. For exmple, if you declare a string INSTRING, then the two word descriptor:

INSTR. : <first word>
INSTRI : <second word>

More about string descriptors on page 107.

BLOCKS The first word of the first executable statement of every block or compound statement which has been given a name is given a label created in the same way as those for arrays above. This label cannot be gone to in the source program. It causes no program inefficiency. This label points at the first word of the compound tail -- not the first word of code generated for the block (skips any procedure or array declaration code).

START The first word of code generated for any given program is given the name "S.".

WARNINGS
Since only the first 6 characters of an identifier are available, it is wise to declare symbols which will be examined by DDRAID in such a way that these six characters will uniquely identify them.

APPENDIX 1

TYPE CONVERSION

The data type BOOLEAN is identical to the data type INTEGER with the following conventions: FALSE = 0 and TRUE ≠ 0.

From \ To	INTEGER	REAL	STRING
INTEGER		Left justify and raise to appropriate power. 1345→1.345@3 -678→-6.78@2	The right 7 bits are converted to a 1 character string with that ASCII code. 48 → "0"
REAL	Drop decimal fractions. 1.345@2→134 -6.7999@1→-67 2.3@-2→ 0		Convert to integer then convert to string. 4.8@1 → "0" 4.899@1 → "0"
STRING	The ASCII code for the first character of string. "0SUM"→ 48 NULL → 0	Convert to integer then to real. "0SUM"→ 4.8@1 NULL → 0	

NOTES: The NULL string is converted to 0, but 0 is converted to the one character string with the ASCII code of 0. If the absolute value of an Integer is greater than 1342 17728, **then** some low order significance will be lost in the conversion to real; otherwise, conversion to real and **then** back to integer will result in the same integer value. If a real number has magnitude greater than 134217728, then conversion to integer will produce an invalid result.

Conversion from real to integer can be sped by a factor of 8 if SHORT reals and integers are used. It is only necessary that one of the data types be SHORT: both the number to be converted and the variable need not be SHORT. SHORTness is a dominate quality in algebraic binary operations. That is, the sum of a SHORT real and a regular real will be treated as a SHORT real. SHORT integers and reals must have an absolute magnitude of less than -134217728

The binary arithmetic, logical, and String operations which follow will accept combinations of arguments of any algebraic types. The type of the result of such an operation is sometimes dependent on the type of its arguments and sometimes fixed. An argument may be converted to a different algebraic type before the operation is performed. The following table describes the results of the arithmetic and logical operations given various combinations of Real and Integer inputs. ARG1 and ARG2 represent the types of the actual arguments (strings go to integers first). ARG1' and ARG2' represent the types of the arguments after any necessary conversions have been made.

OPERATION	ARG1	ARG2	ARG1'	ARG2'	RESULT
+ -	INT	INT	INT	INT	INT*
* ↑ %	REAL	INT	REAL	REAL	REAL
MAX MIN	INT	REAL	REAL	REAL	REAL
	REAL	REAL	REAL	REAL	REAL
LAND LOR	INT	INT	INT	INT	INT
EQV XOR	REAL	INT	REAL	INT	REAL
	INT	REAL	INT	REAL	INT
	REAL	REAL	REAL	REAL	REAL
LSH ROT	INT	INT	INT	INT	INT
	REAL	INT	REAL	INT	REAL
	INT	REAL	INT	INT	INT
	REAL	REAL	REAL	INT	REAL
/	INT	INT	REAL	REAL	REAL
	REAL	INT	REAL	REAL	REAL
	INT	REAL	REAL	REAL	REAL
	REAL	REAL	REAL	REAL	REAL
MOD DIV	INT	INT	INT	INT	INT
	REAL	INT	INT	INT	INT
	INT	REAL	INT	INT	INT
	REAL	REAL	INT	INT	INT

* If ARG2 is negative for the operator "↑", the result is real.

APPENDIX 2

SAIL RESERVED WORDS

ABS ALL AND APPLY ARRAY ARRAY-PDL ASSIGN ASSOC BBPP BEGIN BOOLEAN CASE CASEC CAUSE COMMENT CONTEXT CONTINUE COP CVI CVLIST CVN CVSET DATUM DEFINE DELETE DELIMITERS DIV DO DOC DONE DPB ELSE ELSEC END ENDC ENTRY EQV ERASE EXTERNAL FAIL FALSE FIRST FOR FORC FORLC FOREACH FORGET FORTRAN FORWARD FROM GEQ GLOBAL GO GOT0 IBP IDPB IF IFC ILDB IN INF INITIALIZATION INTEGER INTER INTERNAL INTERROGATE ISTRIPLE ITEM ITEMVAR LABEL LAND LDB LENGTH LEQ LET LIBRARY LOAD-MODULE LOCATION LNOT LOP LOR LSH MAKE MATCHING MAX MEMORY MESSAGE MIN MOD NEEDNEXT NEQ NEXT NEW NEW-ITEMS NOT NOW-SAFE NOW-UNSAFE NULL NULL-CONTEXT NULL-DELIMITERS OF OFC OR OWN PHI PNAMES PRELOAD_WITH PROCEDURE PROTECT-ACS PUT QUICK-CODE REAL RECURSIVE REFERENCE REMEMBER REMOVE REPLACE-DELIMITERS REQUIRE RESTORE RETURN ROT SAFE SECOND SEGMENT-NAME SEGMENT-FILE SET SETC SETO SHORT SIMPLE SPROUT START-CODE STEP STEPC STRING STRING-PDL STRING-SPACE SOURCE-FILE SUCCEED SUCH SWAP SYSTEM-PDL THAT THEN THENC THIRD TO TRUE UNSTACK-DELIMITERS UNTIL UNTILC VALUE VERSION WHILE WHILEC XOR

APPENDIX 3

SAIL PRE-DECLARED **IDENTIFIERS**

ARRBLT ARRINFO ARRTRAN ARRYIN ARRYOUT BACKUP BINDIT BREAKSET CALL CLOSE CLOSIN CLOSO CLRBUF CODE CVASC CVD CVE CVF CVFIL CVG CVIS CVO CVOS CVS CVSI CVSIX CVSTR CVXSTR ENTER EQU EVENT-TYPE FILEINFO GETCHAN GETFORMAT INCHRW INCHRS INCHSL INCHWL INSTR INSTRL INSTRS INPUT INTIN INTSCAN LINOUT LODED LOOKUP MAINPI MTAPE OPEN OUT OUTCHR OUTSTR PTCHRS PTCHRW PTIFRE PTOCNT PTOCHS PTOCHW PTOSTR PTYALL PTYGET PTYIN PTYREL PTYSTR REALIN REALSCAN RELEASE RENAME SCAN SETBREAK SETFORMAT STRBRK TTYIN TTYINL TTYINS WORDIN WORDOUT USERCON USERERR USETI USETO

APPENDIX 4

CHARACTER-IDENTIFIER **EQUIVALENCES**

CHARACTER	RESERVED WORD
∧	AND
≡	EQV
	NOT
	OR
⊗	XOR
∞	INF
	IN
	SUCH THAT
≠	NEQ
≤	LEQ
≥	GEQ
{	SETO
}	SETC
	UNION
	INTER
	ASSOC
↔	SWAP
	!

APPENDIX 5

PARAMETERS TO THE OPEN FUNCTION

```
OPEN (CHANNEL, "DEVICE", MODE,
      INBUFS, OUTBUFS, @COUNT,
      @BRCHAR, @EOF );
```

CHANNEL	System Data Channel, Ø-'17
DEVICE	string giving device name
MODE	data mode, bits 18-21, 23 **enable** error returns
INBUFS	number of input buffers, lh buffer size if ≠Ø
OUTBUFS	number of output buffers
COUNT	text input count (reference)
BRCHAR	break char variable (reference)
EOF	end-of-file and IO error flag (reference)

APPENDIX 6

BREAKSET MODES

I	(Inclusion) string is set of break chars
X	(eXclusion) string of all non-break chars
0	(Omit) string of characters to be omitted from result
S	(skip) break char appears only in BRCHAR variable
A	(Append) break char is last char of result string
R	(Retain) break char is first char of next string
P	(Pass) line numbers appear in input without warning
N	(No numbers) line numbers and the tabs that follow them are removed.
L	(Line no break) line numbers cause input break. BRCHAR is negative. Next input gets line no characters.
E	(Erman) line numbers cause input **break.** Negated line no returned in BRCHAR. Line no removed from input.
D	(Display) after this appears, each line no is listed on the display (if TTY is a DPY) **as** it is dealt with.

APPENDIX 7

MTAPE COMMANDS

MODE	FUNCTION
"A"	Advance past one tape mark (or file)
"B"	Backspace past one tape mark
"F"	Advance one record
"R"	Backspace one record
"W"	Rewind tape
"E"	Write tape mark
"U"	Rewind and unload

APPENDIX 8

COMPILE SWITCHES

D	double size of define pushdown stack
numL	listing control -- num>Ø becomes listing starting addr. num=-1 starts listing after current DDT size. num=-2 starts listing after current RAID size.
P	double size of system pushdown list
Q	double size of string pushdown list
H	for making programs sharable (high segment).
K	for insertion of programs counters.
R	double size of parse pushdown list
numS	set size of string space to num
C	create CREF (cross-reference) input file.
numF	enable various listing formats.

APPENDIX 9

VALID RESPONSES TO ERROR MESSAGES

ACTION RESPONSES

cr	(carriage return) try to continue
C	same as cr
lf	(line feed) continue automatically -- don't stop for user go-ahead after each message
A	same as lf
S	restart
X	exit -- close all files, return to monitor
E	edit. Follow by CR to get file the compiler is working on (or last thing edited, for Runtime routines). Follow with <name> CR to edit <name>.
T	Ditto E only with the TV editor.
D	go to DDT or RAID
B	go to compiler or runtime debugger.

MODES (reset using - e.g. -Q or -L)

K	keep type-ahead (i.e. don't flush)
Q	quiet - turns off display of errors
L	logging - send errors to .LOG file
F	file - ditto L but send to a file specified by <file name> after F
N	Numbers - display "CALLED FROM xxxx"

APPENDIX 10

ERROR CODES

ROUTINE	LOCATION	CONDITIONS, CODE VALUES
CALL	_SKIP_	set TRUE if the UUO skips, FALSE otherwise
CODE	_SKIP_	set TRUE if the constructed instruction skips, FALSE otherwise
CVFIL	_SKIP_	set TRUE if the file input is invalidly specified (wrong punctuation, order, etc.), FALSE otherwise.
CVIS	FLAG param	Set TRUE if no PNAME exists for this Item, FALSE if CVIS succeeds.
CVSI	FLAG param	Set TRUE if no Item exists with this String as PNAME, FALSE if CVSI succeeds.
ENTER	FLAG param	Set FALSE if the ENTER succeeds. Otherwise, the left half is made - 1. Then if the file name was invalid, the right half is made '10. Otherwise it is set to some code from 0 to 7, depending on the type of ENTER failure. These codes are the same as the ENTER UUO codes in [Moorer]. If error '10 (invalid spec.) is returned, an error message (non-fatal) will also be printed, unless you are enabled for user handling of this error (see I/O below).
GETCHAN	result	<Ø if no channel is available.
INCHRS	result	<Ø if no characters are waiting.
INCHSL	FLAG param	≠Ø if no characters are waiting.
INSTRS	FLAG param	≠Ø if no characters are waiting.
I/O	EOF vbl.	0 if no exceptional conditions occurred in an I/O operation. Otherwise, the left half has certain bits turned on, indicating the error: 400000 is a catchall -- improper mode. 200000 means parity error occurred. 100000 means a data error occurred. 40000 means "Record number out of bounds". 20000 means End of File (input only), You are always enabled for bit 20000 (EOF). However, to be allowed to handle any of the others, you must turn on the corresponding bit in the right half of the MODE word in the OPEN for this channel. In addition, the 10000 bit is used to enable user handling of invalid file specifications to ENTER, LOOKUP, and RENAME (see above). '7500017 in the MODE parameter would enable a dump mode file for user handling of ALL I/O errors on this channel. If you are not enabled for a given error, an error message (which may or may not be fatal) will be printed, and the error code word set as indicated. In addition, the number of words actually transferred is stored in the right half of this variable for ARRY IN, ARRYOUT.
LOOKUP	FLAG param	Same as ENTER.
OPEN	EOF vbl	If 0 on entry, prints fatal error message if OPEN fails. If ≠Ø on entry, always returns to user -- still ≠Ø if OPEN failed, 0 if it succeeded.
RENAME	FLAG param	Same as ENTER.
TTYINS	FLAG param	Same as INSTRS.
Substrings	_SKIP_	Consider ST[X TO Y]. If Y>LENGTH(ST) it is set to LENGTH(ST) and rh(_SKIP_) is made -1. If X<1 it is set to 1. If X>Y it is set to Y+1 (guaranteeing a null String result). In either case, lh(_SKIP_) is set to -1. The ST[X FOR Y] case is first converted to the other case, then executed.

You should also refer to the table for Input ,page 35, describing the various combinations of the BRCHAR and EOF variables and their meanings.

APPENDIX 11

INDICES FOR INTERRUPTS

STANFORD INTERRUPT SYSTEM

NAME	NUMBER	DESCRIPTION
INTSWW_INX	0	You will receive an Interrupt when your job is about to be swapped out.
INTSWD_INX	1	You will receive an interrupt when your job is swapped back into core If you are activated for Interrupts for swap out also, you will receive these two interrupts as a pair in **the** expected order every time your job is swapped.
INTSHW_INX	2	You will receive an interrupt when your job is about to be shuffled.
INTSHD_INX	3	You will receive an Interrupt when your job has been shuffled.
INTTTY_INX	4	You will receive an Interrupt every time your program would be activated due to the teletype if it were waiting for the teletype. As long as you do not ask for more than there is in the teletype buffer, you may read from the teletype at interrupt level.
INTPTO_INX	5	You will be interrupted every time the PTY job goes into a wait state waiting tor you to sent it characters
INTMAIL_INX	6	Interrupts whenever someone SENDs you mail (see [Moorer], section II D.17) You may read the letter at Interrupt level.
INTPTI_INX	8	You will be interrupted everytime any job on a PTY you own send you 3 character (or line).
INTPAR_INX	9	interrupts you on parity errors in your core image
INTCLK_INX	10	You will be interrupted at every clock tick (1/60th of a second).
INTINR_INX	11	IMP Interrupt by receiver
INTINS_INX	12	IMP interrupt by sender.
INTIMS_INX	13	IMP status change interrupt.
INTINP_INX	14	IMP input waiting
INTTTI_INX	15	You will be interrupted whenever <esc> I is typed on your teletype.
INTPOV_INX	19	Interrupts you on push-down overflow.
INTILM_INX	22	Interrupts you on illegal memory references, that is, references to memory outside of your core image
INTNXM_INX	23	You will receive an interrupt whenever your program references non-existant memory.
INTFOV_INX	29	Interrupts you on floating overflow.
INTOV_INX	32	Interrupts you on arithmetic overflow.

Bits 33 through 36 are left to the user. REQUIRE "SYS:PROCES.DEF" SOURCE-FILE to define the above names, NOTE: to program yourself for more than one Interrupt, you must execute two separate INTMAP statements.

EXPORT SAIL INTERRUPT SYSTEM

NAME	NUMBER	DESCRIPTION
INTPOV_APR	19	Interrupts you on push-down stack overflow
INTILM_APR	22	Interrupts you on illegal memory references that is, references to memory outside of your core Image.
INTNXM_APR	23	You will receive an interrupt whenever your program references non-existant memory.
INTFOV_APR	29	Interrupts you on floating overflow
INTOV_APR	32	Interrupts you on arithmetic overflow

APPENDIX 12

BIT NAMES FOR PROCESS CONSTRUCTS

SPROUT OPTIONS

BITS	NAME	DESCRIPTION
14-17	QUANTUM(X)	Q ← IF X=0 THEN 4 ELSE 2TX; The process will be given a quantum of Q clock ticks, indicating that if the user is using CLKMOD to handle clock interrupts, the process should be run for at most Q clock ticks, before calling the scheduler. (see about CLKMOD, page 79 for details on making processes "time share").
18-21	STRINGSTACK(X)	S ← IF X-0 THEN 16 ELSE X*32; The process will be given S words of string stack.
22-27	PSTACK(X)	P←IF X=0 THEN 32 ELSE X*32; The process will be given P words of arithmetic stack.
28-31	PRIORITY(X)	P ← IF X=0 THEN 7 ELSE X; The process will be given a priority of P. 0 is the highest priority, and reserved for the SAIL system. 15 is the lowest priority. Priorities determine which ready process the scheduler will next pick to make running.
32	SUSPHIM	If set, suspend the newly sprouted process.
33		Not used at present.
34	SUSPME	If set, suspend the process in which this sprout statement occurs.
35	RUNME	If set, continue to run the process in which this sprout statement occurs.

RESUME OPTIONS

BITS	NAME	DESCRIPTION
33-32	READYME	If 33-32 is 1, then the current process will not be suspended, but be made ready.
	KILLME	If 33-32 IS 2; then the current process will be terminated.
	IRUN	If 33-32 is 3, then the current process will not be suspended, but be made running. The newly resumed process will be made ready.
34		This should always be zero.
35	NOTNOW	If set, this bit makes the newly resumed process ready instead of running. If 33-32 are not 3, then this bit causes a rescheduling.

CAUSE OPTIONS

BITS	NAME	DESCRIPTION
35	DONTSAVE	Never put the <event item> on the notice queue. If there is no process on the wait queue, this makes the cause statement a no-op.
34	TELLALL	Wake all processes waiting for this event. Give them all this item. The highest priority process will be made running, others will be made ready.
33	RESCHEDULE	Reschedule as soon as possible (i.e. immediately after the cause procedure has completed executed).

INTERROGATE OPTIONS

BITS	NAME	DESCRIPTION
35	RETAIN	Leave the event notice on the notice queue, but still return the notice as the value of the interrogate. If the process goes into a wait state as a result of this Interrogate, and is subsequently awakened by a Cause statement, then the DONTSAVE bit in the Cause statement will over ride the RETAIN bit in the Interrogate if both are on.
34	WAIT	If the notice queue is empty, then suspend the process executing the interrogate and put its process item on the wait queue.
33	RESCHEDULE	Reschedule as soon as possible (i.e. immediately after execution of the interrogate procedure).
32	SAY-WHICH	Creates the association

EVENT-TYPE ⊗ <event notice>≡ <event type> where <event type> is the type of the event returned. Useful with the set form of the Interrogate construct.

APPENDIX 13

STATEMENT COUNTER SYSTEM

GENERAL DISCUSSION

The new SAIL compiler contains a feature which allows you to determine conveniently the frequency of execution of each statement in your SAIL program.

This is accomplished by inserting an array of counters and placing AOS instructions at various points in the object program (such as in loops and conditional statements). A routine is called to zero the counter array before your program is entered and another routine is called to write out the array before calling EXIT.

Since not all programs exit in the normal fashion (i.e. falling out the bottom), it is possible to call either the zero routine or the output routine as an EXTERNAL PROCEDURE.

Another program, called PROFIL, is used to merge the listing file produced by the SAIL compiler with the file of counters produced by the execution run of your program. The output of the PROFIL program is an indented listing of your SAIL program with execution counts in the right hand margin. The output format of PROFIL is reasonably flexible, with several "switches" to control it.

Since the AOS instructions access fixed locations, and they are placed only where needed to determine program flow, they should not add much overhead to the execution time. Although no large study has been made, the counters seem to contribute about 2% to the execution time of the profile program, which has a fairly deeply nested structure.

SAIL EXTENSION

The mechanism for inserting counters is controlled by a compiler switch. To tell the compiler to insert counters, you give it a /K switch. (/C was already used for something else.) It is also necessary to produce a listing file, since the PROFIL program needs it. In fact, the /K switch is ignored unless a listing is called for. Specifying /K has several effects on the listing. First, macros are expanded and macro names not listed. This is necessary so that PROFIL will know about block structure, etc. Also, the listing of PC and line numbers is suppressed. The current version of PROFIL is confused by all those numbers and anyway, the lines of the PROFIL listing can differ somewhat from the lines of the original source. The final change in the listing is the inclusion of markers telling where counters have been inserted. Most of these are Ignored by the present PROFIL since it is smart enough to know where they are from the

program context. The ones that it does use are the markers for counters inserted into conditional and case expressions.

At the end of each program (i.e. each separate compilation) is the block of counters, preceeded by a small data block used by the zero and output routines. This block contains such information as the number of counters, the name of the list file, and a link to other such blocks of counters. The first counter location is given omaname .KOUNT, which is accessible from RAID, but cannot be referenced by the SAIL program itself.

The routine K.ZERO is called to zero the counters. If for some reason you wish to zero them yourself, (like if you're only interested in steady state execution counts) you can reference this routine by including the declaration:

EXTERNAL PROCEDURE K-ZERO;

The outputting of the counters is done by the routine K.OUT. It uses the SAIL routine GETCHAN to find a spare channel, does a single dump mode output which writes out all the counters for all the programs loaded having counters, and then releases the channel. The file which it writes is xxx.KNT, where xxx is the name of the list file of the first program loaded having counters (usually the name of the SAIL source file). If there are no counters, K.OUT simply returns, This routine can also be referenced by including the declaration:

EXTERNAL PROCEDURE K-OUT;

PROFILE PROGRAM

The program PROFIL is used to produce the program profile, i.e. the listing complete with statement counts. It operates in the following manner. First it reads in the file xxx.KNT created by the execution of the user program. This file contains the values of the counters and the names of the list files of the programs loaded which had counters. It then reads the the list files and produces the profile.

The format of the listing is such that only statements executed the same number of times are listed on a single line. In the case of conditional statements, the statement is continued on a new line after the word THEN. Conditional expressions and case expression, on the other hand, are still listed on a single line. In order that you might know the execution counts, they are inserted into the text surrounded by two "brokets" (e.g. <<15>>).

PROFIL expects a command string of the standard form for CUSP's, i.e.

<output>←<input> {switches}.

where the <Input> is the name of the .KNT is assumed. If the output device is the DSK, the output file will have a default extension of .PFL. Although the line spacing will probably be different from the source, PROFIL makes an effort to keep any page spacing that was in the source. here are several possibilities for switches, for which the pertinent ones are:

```
/nB   Indent n spaces for blocks (default 4)
/nC   Indent n spaces for continuations (default 2)
/F    Fill out every 4th line with "..." (default ON)
/I    Ignore comments, strip them from the listing
/nK   Make counter array of size n (default 200)
/nL   Maximum line length of n (default 120)
/N    Suppress /F feature
/S    Stop after this profile
/T    TTY mode = /1C/2B/F/80L
```

SAMPLE RUN

Suppose that you have a SAIL program named FOO.SAI for which you desire a profile. The following statements will give you one.

```
.EX /LIST FOO(K)  (or TRY or DEB or what have you)
.''any input to FOO...

EXIT

↑C
.R PROFIL
*FOO←FOO/T/S

EXIT

↑C
.  At this point, the file FOO.PFL contains the profile,
   suitable for typing on the TTY or editing.
```

APPENDIX 14

ARRAY IMPLEMENTATION

Let STRINGAR be 1 (TRUE) if the array in question is a String array, 0 (FALSE) otherwise. Then a SAIL array of n dimensions has the following format:

```
HEAD:    →DATAWD        ;→ MEANS "POINTS AT"
         HEAD-END-1
ARRHED:  BASE_WORD      ;SEE BELOW
         LOWER_BD(n)
         UPPER_BD(n)
         MULT(n)
         . . .
         LOWER_BD(1)
         UPPER_BD(1)
         MULT(1)
         NUM_DIMS,,TOTAL_SIZE
DATAWD:  BLOCK TOTAL_SIZE
         <sometimes a few extra words>
END:     400000,,→HEAD
```

HEAD The first two words of each array, and the last, are control words for the dynamic storage allocator. These words are always present for an array. The array access code does not refer to them.

ARRHED Each array is preceded by a block of 3*n+2 control words. The BASE-WORD entry is explained later.

NUM_DIMS This is the dimensionality of the array. If STRINGAR, this value is negated before storage in the left half.

DATAWD This is stored in the core location bearing the name of the array (see symbols, page 97). If it is a string array, DATAWD+1 is stored instead.

TOTAL-SIZE The total number of accessible elements (double if STRINGAR) in the array.

BOUNDS The lower bound and upper bound for each dimension are stored in this table, the left-hand index values occupying the higher addresses (closest to the array data). If they are constants, the compiler will remember them too and try for better code (i.e. immediate operands).

MULT This number, for dimension m, is the product of the total number of elements of dimensions m+1 through n. MULT for the last dimension is always 1.

BASE-WORD This is DATAWD minus the sum of (STRINGAR+1) * LOWER_BD(m) * MULT(m) for all m from 1 to n.

The formula for calculating the address of A[I,J,K] is:

```
address(A[I,J,K]) =
    address(DATAWD) +
      (I-LOWER_BD(1))*MULT(1) +
      (J-LOWER_BD(2))*MULT(2) +
      (K-LOWER_BD(3))
```

This expands to

```
address(A[I,J,K]) =
   address(DATAWD) +
     I*MULT(1) + J*MULT(2) + K
      -(LOWER_BD(1)*MULT(1) +
          LOWER_BD(2)*MULT(2) +
             LOWER_BD(3)
```

which is

```
BASE_WORD + I*MULT(1) + J*MULT(2) + K.
```

By pre-calculating the effects of the lower bounds, several instructions are saved for each array reference.

APPENDIX 15

STRING IMPLEMENTATION

STRING DESCRIPTORS

A SAIL String has two distinct parts: the descriptor and the text. The descriptor is unique and has the following format:

WORD 1: CONST,,LENGTH
WORD2: BYTP

1) CONST. This entry is 0 if the String is a constant (the descriptor will not be altered, and the String text is not in String space, is therefore not subject to garbage collection), and non-zero otherwise.

2) LENGTH. This number is zero for any null String; otherwise it is the number of text characters.

3) BYTP. If LENGTH is 0, this byte pointer is never checked (it need not even be a valid byte pointer. Otherwise, an ILDB machine instruction pointed at the BYTP word will retrieve the first text character of the String. The text for a String may begin at any point in a word. The characters are stored as LENGTH cont iguous characters.

A SAIL String variable contains the two word descriptor for that variable. The identifier naming it points to WORD1 of that descriptor. If a String is declared INTERNAL, a symbol is formed to reference WORD2 by taking all characters from the original name (up to 5) and concatenating a "." (OUTSTRING's second word would be labeled OUTST.).

When a String is passed by reference to a procedure, the address of WORD2 is placed in the P-stack (see page 107). For VALUE Strings both descriptor words are pushed onto the SP stack.

A String array is a block of 2-word String descriptors. The array descriptor (see page 1Ø6) points at the second word of the first descriptor in the array.

Information is generated by the compiler to allow the locations of all non-constant strings to be found for purposes of garbage-collection and initialization , All String variables and arrays are cleared to NULL whenever a SAIL program is started or restarted.

APPENDIX 16

PROCEDURE IMPLEMENTATION

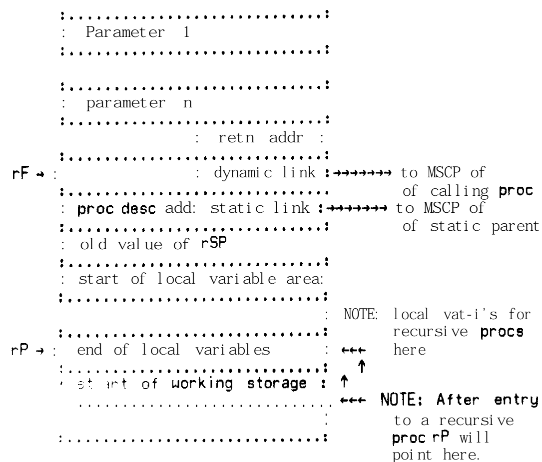
A VERY IMPORTANT NOTE

When a procedure is entered, it places three words of control Information on the run time (P) stack. This "mark stack control packet" (MSCP) contains, among other things, pointers to the control packets for the procedure's dynamic and static parents. Also, register rF (register '12) is set to point at this area. This pointer is then used to access procedure parameters and other "in stack" objects, such as the local variables of a recursive procedure. Also, many of the run-time routines (including the string garbage collector) use rF to find various bits and pieces of vital information. Therefore, THE USER MUST NOT HARM REGISTER '12. In particular, one should not call any runtime routines, ask the compiler to access any stacked variables, or exit any blocks with this register changed from the value given it by SAIL. If you wish to refer in assembly language to a procedure parameter, or the like, the safest way is name it, and let SAIL do the address arithmetic. (Similarly one may use the Access construct).

Most of the remainder of this section may probably be ignored by the occasional user of assembly language who follows the advice we have just given. It is being included for the benefit of those users whose use of "hand coded" routines is sufficiently frequent (or sufficiently hairy) that they need to know more about the way procedures work.

STACK FRAME

Shown here is the stack frame of a recursive procedure



If a formal parameter is a value parameter, then the actual parameter value is kept on the stack. If a formal parameter is a reference parameter, then the

address of the actual parameter is put on the stack. Non-own string locals (to recursive procedures) and string value parameters are kept on the string (SP= '16) stack. The stack frame for a non-recursive procedure is the same except that there are no local variables on the stack. The stack frame for a SIMPLE procedure consists only of the parameters and the return address.

ACCESSING THINGS ON THE STACK

SIMPLE procedures access their parameters relative to the top-of-stack pointers SP(for strings) and P (for everything else). Thus the the k'th (of n) string value parameter would be accessed by

```
OP      AC,2*k-2*n(SP)  ;(SP='16)
```

and the j'th (of m) "arithmetic" -- ie not value string -- parameter would be accessed by

```
OP      AC, j-m-1(P)    ;(P='17)
```

Non-SIMPLE procedures use rF (register '12) as a base for addressing parameters and recursive locals. Thus the j'th parameter-would be accessed by

```
OP      AC,j-m-2(rF)
```

or, in the case of a string, by

```
MOVE    ACX,2(rF)         ;points at top of
                          ;string stack uhen
                          ;proc was entered
OP      ACY,2*k-2*m(ACX)
```

Similarly, recursive locals are addressed using positive displacements from rF.

An up-level reference to a procedure's parent is made by executing the instruction

```
HRRZ    AC,1(rF)          ;now AC points at
                          ;stack frame of parent
```

and then using AC in the place of rF in the access sequences above, iterating the process if need be to get at one's grandparent, or some more distant lexical ancestor.

NOTE: When SAIL compiled code needs to make such an up-level reference it keeps track of any Intermediate registers (called "display" registers) that may have been loaded. Thus, if you use several up-level references together, you only pay once for setting up the "display", unless some intervening procedure call or the like should cause SAIL to forget whatever was in its accumulators. Note here that if a display register is thrown away, there is no attempt to save its value. At some future date this may be done. It was felt, however, that the minimal (usually zero) gain in speed was just not worth the extra hair that this would entail

ACTIONS IN THE PROLOGUE FOR NON-SIMPLE PROCEDURES

The algorithm given here is that for a recursive procedure being declared inside another procedure. The examples show how it is simplified when possible.

1. Pick up proc descriptor address.

2. Push old rF onto the stack.

3. Calculate static link. (a). Must loop back through the static links to grab it. (b). once calculated put together with the PDA and put it on the stack.

4. Push current rSP onto the stack.

5. Increment stack past locals & check for overflow.

6. Zero out whatever you have to.

7. Set rF to point at the MSCP.

EXAMPLES:

1. A non-recursive entry (note: in this section only case! where F is needed are considered.

```
PUSH    P,rF              ;SAVE DYNAMIC LINK
SKIPA   AC,rF
MOVE    AC,1(AC)          ;GO UP STATIC LINK
HLRZ    TEMP,1(AC)        ;LOOK AT POA IN STACK
CAIE    TEMP,PPDA         ;IS IT THE SAME AS PARENTS
JRST    .-3               ;NO
HRLI    AC,PDA            ;PICK UP PROC OESC
PUSH    P,AC              ;SAVE STATIC LINK
PUSH    P,SP
HRRZI   rF,-2(P)          ;NEW RF
```

In the case that the procedure is declared in the outer block we don't need to worry about the static link and the prologue can look like

```
PUSH    P,rF                      ;SAVE DYNAMIC LINK
PUSH    P,[XWD PDA,0]             ;STATIC LINK WORD
PUSH    P,SP                      ;SAVE STRING STACK
HRRZI   rF,-2(P)                  ;NEW F REGISTER
```

2. Recursive entry -- i.e one with locals in the stack.

```
PUSH    P,rF              ;SAVE OYNAMIC LINK
SKIPA   AC,rF
MOVE    AC,1(AC)          ;GO UP STATIC LINK
HLRZ    TEMP, (AC)        ;LOOK AT POA IN STACK
CAIE    TEMP,PPDA         ;IS IT THE SAME AS PARENTS
JRST    .-3               ;NO
HRLI    AC,PDA            ;PICK UP PROC OESC
PUSH    P,AC              ;SAVE STATIC LINK
PUSH    P,SP
HRLZI   TEMP,1(P)
HRRI    TEMP,2(P)
ADD     P,[XWD locals, locals1 ;create space for
CAIL    P,0                    :arith locals
<trigger pdl ov error>
SETZM   -1(TEMP)          ;zero out locals
HLT     TEMP,(P)
HRLZI   TEMP,1(SP)
HRRI    TEMP.2(SP)
ADD     SP,[XWD 2* string locals,2* string locals3
CAIL    SP,0              ;check for pdl ov
<cause pdl cv error>
SETZM   -1(TEMP)
BLT     TEMP, (SP)        ;zero out string locals
```

```
HRRZI   rF,- locals-3(P)
```

The BLT of zeros is replaced by repeated pushes of zero if there are only a few locals. Again, the loop is replaced by a simple push if the procedure is declared in the outer block.

ACTIONS AT THE EPILOGUE FOR NON-SIMPLE PROCEDURES

1. If returning a value, set it into 1 or onto right spot in the string stack.

2. Do any deallocataions that need to be made.

4. Restore rF.

5. Roll back stack.

6. Return either via POPJ P, or by JRST @mumble(P)

EXAMPLES:

1. No parameters.

```
<step 1>
<step 2>
MOVE    rF,(rF)
SUB     P,[XWD M+3,M+3]     ;M= # LOCAL VARS
POPJ    P,
```

2. n string parameters, m other parameters, k string locals on stack, j other locals on stack.

```
<step 1>
<step 2>
MOVE    rF,(rF)
SUB     SP,[XWD 2*k+2*n,2*k+2*n]
SUB     P,[XWD jtmt3, j+m+3]     ;POPS THE STACK
JRST    @M+1(P)
```

SIMPLE procedures are similar, except that rF is never changed.

PROCEDURE DESCRIPTORS

Procedure descriptors are used by the storage allocation system, the interpretive caller, a planned debugger, and various other parts of SAIL. They are not put out for SIMPLE procedures. The entries are shown as they are at the present time. No promise is made that they will not be different tomorrow. If you do not understand this page, do not worry too much about it

```
        link for pd list
0:      entry address
        string pointer for
2:      procedure id
3:      type info for procedure,,0
4:      string params*2,, arith params+1
5:      +ss displ,,+ as displ
6:      lexic lev,,→→local var info
7:      display level,,→→proc param stuff
10:     pda,,0
11:     pcnt at end of mksemt,,parent's pda
12:     pcnt at prdec,,loc for jrst exit
13:     type info for first argument,,0

        type info for last argument,,0
lvi:    byte (4)type(9)lexical-level(23)location
```

The type codes in the lvi (local variable infor) block are as follows:

```
type = 0     end of procedure area
type = 1     arith array
type = 2     string array
type = 3     set or list
type = 4     set array
type = 5     foreach search control block
type = 6     list of all processes dependent on
             this block.
type = 7     context
type = 10    a cleanup to be executed
type = 17    block boundary. Location gives base
             location of parents block's information.
```

local variable info for each block is organized as

```
info for var

info for var
17,lev,loc of parent block bbw
```

REFERENCES

Feldman — Feldman, J.A. and Rovner, PD. An Algal-Based Associative Language, Comm. ACM 12, 8 (Aug. 1969), 439-449.

J.A. Feldman, J.R. Low, DC. Swinehart, and R.H. Taylor. Recent Developments in SAIL. AFIPS Fall Joint Conference, 1972, 1193-1202

Frost — Frost, M. UUO Manual Stanford A-I Laboratory Operating Note 55.3 (June 1973) supersedes Moorer (below)

Moorer — Moorer, J.A. Stanford A-I Project Monitor Manual, Sailons 54 and 55 (Sep. 1969).

Petit — Petit, P. RAID Manual, Sailon 58.1, (Feb. 197 1)

Savitzky — Savitzky, S.R. Son of Stopgap, Sailon 50.1, (Sep. 1969), a revision of Stopgap, Sailon 50, by W.F. Weiher.

Swinehart & Sproull — Swinehart, DC. and Sproull, R.F. SAIL, SAILON 57.2 (Jan. 1971), second of three versions of the Sail manual.

Weiher — Weiher, W.F. Loader Input Format, Sailon 46 (Oct. 1968).

INDEX

∧ (AND) 23
¬ (NOT) 23
∞ in substrings 24
∞, in list REMOVEs 56
∩ (INTERSECTION) 65
∪ (UNION) 65
∨ (OR) 23
% (integer or real division) 24
& (CONCATENATION), of strings 24
&, of lists 65
-, of sets 65
/ (real divison) 24
<>≤≥=≠ (RELATIONS) 23
?, Foreach itemvars 59
?, in Binding Booleans 58
?, Matching procedure formals 61
<algebraic-expression> 20
<apply-construct> 76
<arg_list_specifier> 76
<array_declaration> 3
<array-list> 3
<array_type> 51
<assign-statement> 76
<assignc> 45
<assignment-expression> 20
<assignment-statement> 13
<associative-statement> 55
<backtracking-statement> 29
<binding-list> 55
<block> 1
<boolean-expression> 20
<case-expression> 20
<case-statement> 13
<cause-statement> 72
<cleanup-declaration> 4
<code-block> 26
<command-line> 91
<compound-statement> 1
<cond_comp_statement> 45
<conditional-expression> 20
<conditional-statement> 13
<context-declaration> 29
<context-element> 29
<declaration> 3, 51
<define> 45
<derived-set> 63
<do-statement> 13
<element-list> 55

<element> 55
<element>, Foreach 59
<event-statement> 7 2
<expression> 20
<for-statement> 13
<foreach_statement> 5 5
<go-to-statement> 13
<id-list> 3
<if_statement> 1 3
<interrogate-construct> 72
<item-expression> 63
<item_primary> 6 3
<item_type> 5 1
<itemvar_type> 5 1
<join-statement> 67
<label-declaration> 3
<leap-expression> 63
<leap-relational> 63
<leap-statement> 55
<list-expression> 63
<list-statement> 55
<macro-body> 45
<macro-call> 45
<preload_specification> 3
<procedure-call> 14
<procedure-declaration> 4, 52
<procedure-head> 4
<procedure-type> 5 2
<process-statement> 67
<ref_item_construct> 7 6
<require-specification> 4
<resume-construct> 67
<safety-statement> 14
<set-expression> 63
<set-statement> 55
<simple_formal_type> 5 2
<simple_type> 5 1
<sprout-statement> 67
<statement> 1
<substring_spec> 2 1
<suc_fail_statement> 5 5
<suspend-statement> 67
<swap-statement> 13
<synonym-declaration> 4
<terminate-statement> 67
<triple> 55
<type-qualifier> 3
<typed-item-expression> 88
<variable> 88
<while-statement> 13
ABS 25
ACCESS 27
algebraic variables 6
allocation of variables and arrays 9
ANSWER 74, 86
ANY 64
ANY, in Binding Boolean 58
ANY, in Derived Sets 58
ANY, in Erase statement 57
ANY, in Foreach 6 1
AOBJN pointer 81
APPLY 76
ARRAY-PDL 9
Array element designation 88
Arrays, allocation 9
Arrays, as parameters 7
Arrays, declaration 6
Arrays, initialization and reinitialization 9
Arrays, outer block 4, 6
Arrays, OWN 6
Arrays, PRELOADed 6
Arrays, SAFE declaration 6
Arrays, storage convention 6
ARRBLT 43
ARRINFO 4 3
ARRTRAN 43
ARRYIN 3 6
ARRYOUT 37
ASH 24
ASKNTC 75, 86
ASSIGN 76
ASSIGNC 5 0
assignment expressions 2 2
Assignment statement, semantics 14
ASSOCIATIONS 53
Associations, ERASE 5 7
Associations, implementation 54
Associations, introduction 5 1
Associations, MAKE 5 7
Associations, searching for 57
associative booleans 65
associative context 59
Associative search 57
Associative search, controling hash 58
associative search, relative speeds 61
associative searches, introduction 5 1
associative store 51, 53
Associative store, searching 57
attribute 58
Backtracking, introduction 29
BACKUP 38
BIND 58
Binding Boolean 58, 65
Binding Booleans, general considerations 58
BINDIT 6 5
BINDIT, in Binding Boolean 58

BINDIT, in Derived Sets 58
BINDIT, in Foreach 6 1
BINDIT, in Foreaches 59
BINDIT, in Matching Procedures 61
Block names 1, 96
Boolean Expression <element> 60
Boolean, declaration 6
bound 58
Bracketed Triple item 57
Bracketed Triple Item Retrieval 57
Bracketed Triple Item retrieval 58
Bracketed Triple item retrieval, general considerat ions 5 8
Bracketed Triple Items, ERASE 57
BREAKSET 3 3
BRKERS 83
BRKMAK 83
BRKOFF 83
BUCKETS 58
BUILT-IN 49
Byte pointers, creation 43
CALL 42
CALLER 85
calling block 80
CASE expressions 22
CASE statement 16
CASEC 49
CAUSE 72
CAUSE, <options> 72, 103
CAUSE, user defined procedures for 74
CAUSE1 74, 86
Causing events, introduction 72
CHECK-TYPE 49
CHECKED 53, 56
Checked, formal parameters 54
CHECKED, in associative searches 58
Checked, itemvar procedures 54
Checked, type checking 65
CLEANUP 9
CLKMOD 79
CLOSE 32
CLOSIN 3 2
CLOSO 32
CLRBUF 38
CODE 42
command line 91
Comment 1
COMMENTS 8 9
compile time expressions 47
concatenation of lists 65
conditional compilation 49
Conditional Statements, ambiguity 15
Constants, arithmetic 89
Constants, octal 89
Constants, real 89
Constants, string 89
constructive item expressions 64
CONTEXT 29
Context elements 30
CONTINUE statement 18
Conversions, algebraic 2 1
COP 64, 85
coroutining with RESUMEs 69
CVASC 41
CVD 41
CVE 40
CVF 40
CVFIL 4 3
CVG 40
CVI 54, 83
CVIS 66, 84
CVLIST 8 3
CVMS 48
CVN 54, 83
CVO 41
CVOS 40
CVS 40
CVSET 83
CVSI 66, 84
CVSIX 4 1
CVSTR 41
CVXSTR 41
DATUM 52, 56, 88
DATUM, type checking 65
DDT 96
deallocation of variables and arrays 9
DECLARATION (a function) 49
DEFINE 45, 46, 48, 49
DEFPRI 6 8
DEFPSS 68
DEFQNT 68
DEFSSS 68
DEL-PNAME 66, 84
DELETE 55, 56
delimited-anything 49
delimited-expr 49
delimited strings 47
Delimiters 46
DELIMITERS 46
DELIMITERS, NULL 46
Delimiters, null 46
DEPENDENTS 68
Derived sets 65
Derived Sets, general considerations 58
DFR1IN 8 1
DFRINT 8 0

DI buffer 8 1
DIV 24
DO statement 16
DONE statement 17
DONTSAVE 72, 103
DPB 42
DRYROT 9 5
ENTER 33
ENTRY specification 11
EQU 41
EQV 23
ERASE 57
ERASE, in a Foreach 61
ERROR-MODES 95
error messages 95
EVALDEFINE 5 0
EVENT-TYPE 73, 104
event notices 72
Event type items, datums of 73
event types 72
Events, introduction 7 2
EXTERNAL declaration 4, 11
EXTERNAL procedures 8, 10
FAIL 55, 56, 61, 69
FALSE, definition 8 9
FILEINFO 4 3
FIRST 57, 84
FOR statement 15
FORC 49
FOREACH 5 5
Foreach <element>, Boolean Expression 60
Foreach <element>, List membership 59
Foreach <element>, Retrieval Triple 60
Foreach <element>, Set membership 59
Foreach <element>s 5 9
Foreach itemvars 59
Foreach searches, relative speeds 61
FOREACH, execution of 59
FOREACH, general considerations 58
FOREACH, increase speed of 58
FOREACH, main discussion of 58
Foreach, Matching Procedure <element> 61
Foreach, satisfiers 59
FORGET 29, 30
FORLC 49
formal parameters, Leap 54
formals 7
FORTRAN procedures 8, 11, 19
FORTRAN, actual parameters 9
FORWARD declaration 4
FORWARD procedures 7
generatton of symbols using macros 47
Gensym 47
GETCHAN 33
GETFORMAT 40
Go To Statements, restrictions 15
GO TO, into a Foreach 59
IBP 42
identifiers 89
IDPB 4 2
IF expressions 22
IF statement 14
IFC 49
IFCR 4 9
ILDB 4 2
ILL MEM REF 90
ILLEGAL UUO 90
IN-CONTEXT 43
INCHRS 3 8
INCHRW 3 8
INCHSL 3 8
INCHWL 3 8
initialization 9
INITIALIZATION 10
inner block 1
INPUT 35
INSTR 3 8
INSTRL 3 8
INSTRS 3 8
INT..._APR 1 0 2
INT..._INX 1 0 2
integer constants 89
Integers, range 5
INTERNAL declaration 4, 11
INTERNAL procedures 8
INTERROGATE 73
INTERROGATE, <opt ions> 73, 103
INTERROGATE, set form of 73
INTERROGATE, user defined procedures for 74
Interrupt codes 102
Interrupts, complicated deferred **80**
INTIN 3 7
INTMAP 8 0
INTPRO 8 1
INTRPT 70, 79
INTSCAN 3 8
INTSET 8 0
INTTBL 8 1
IRUN 70, 103
ITEM 52
item booleans 65
Item, <typed-item-expression> 88
Items & Itemvars, distinction between 53
Items, ANY 64
Items, BINDIT 65
Items, Bracketed Triple 57

items, creation of 52
Items, Datums of 52
Items, declared 52
Items, DELETE 56
Items, implementation 54
Items, internal & external 52
Items, internal &external 54
Items, introduct ion 5 1
Items, NEW 64
Items, Pnames 66
Items, props of 66
Items, scope 52
Items, type checking 65
Items, type of 52
Items, with array datums 52
ITEMVAR 5 3
Itemvars & Items, distinction between 53
Itemvars, CHECKED 53
Itemvars, implementation 54
Itemvars, initialization 53
Itemvars, scope 53
itemvars, type checking 53, 56
Itemvars, types of 53
JOIN 70
KILLME 70, 103
Label use 5
Labels, as actual parameters 9
Labels, restrictions 15
LAND 23
LDB 42
leap booleans 65
Leap, introduct ion 5 1
LENGTH 41, 85
LET 9
letters, legal Sail letters 89
LIBRARY 10
Library, runtime 31
LINOUT 3 6
LIST 53
list booleans 6 5
list element designator 88
List element designators 64
list expressions 65
List membership <element> 59
list, sublists 6 5
Lists, automatic conversion 56
lists, concatenation 65
lists, initialization 65
Lists, PUT 56
Lists,REMOVE 56
LISTX 8 4
LOAD-MODULE 10
LOCATION 25
LODED 38
Logical expressions 23
LOOKUP 33
loop block 17
LOP 41, 64, 84
LOR 23
LPARRAY 49
LSH 24
Macro bodies 47
Macro bodies, concatenation in 48
macro body delimiters 46
macro declarations 46
Macro declarations, scope 46
macro parameter delimiters 46
Macros with parameters 48
Macros without parameters 46
MAKE 55, 57
MAKE, in a Foreach 61
Matching Procedures 6 1
Matching procedures, as processes 69
Matching Procedures, sharing memory 62
MAX 23
MEMORY 25
MESSAGE 50
MIN 23
MKEVTT 72, 85
MOD 24
MTAPE 37
MULTIN 7 5
MYPROC 85
NEEDNEXT 1 7
NEW 63, 64
NEW-ITEMS 64
NEW-PNAME 66, 84
NEXT statement 17
NIL 65
No one to run. 70
NOJOY 7 4
NOMAC 50
NOPOLL 7 0
NOTCQ 74
notice queue 72
NOTNOW 70, 103
NOW-SAFE 19
NOW-UNSAFE 19
NULL-CONTEXT 30
NULL DELIMITERS 46
null delimiters mode 46
NULL, definition 89
object 58
OPEN 31
operator precedence 22
OUT 36

OUTCHR 38
outer block 1
OUTSTR 38
OWN 5
parametric procedures 8
PHI 65
Pnames 66
PNAMES 66
POINT 43
POLL 71
POLLING-INTERVAL 7 1
Polling points 70
PRELOADed arrays 6
Printnames of items 66
PRIORITY(X) 68, 103
PRISET 8 5
Procedure body, emptiness 5
Procedure Calls, actual parameters 18
Procedure Calls, semant ics 18
Procedures, as actual parameters 18
Procedures, assembly language 11
Procedures, declaration 7
Procedures, defaults in declarations 8
procedures, Leap 54
Procedures, parametric 8
Procedures, restrictions 9
Procedures, restrictions on formal parameters 7
Procedures, separately compiled 10
process item 67
process procedure 6 7
Process procedures, Matching 69
Process procedures, recursive 69
Processes, control of scheduling 70
processes, creation of 6 7
Processes, dependency of 68
Processes, inside recursive procedures 68
PROCESSES, introduct ion 6 7
Processes, resumption of 69
Processes, sharable memory 69
Processes, status of 67
Processes, suspension of 69
Processes, termination of 69
Program name, for DDT 1
PROPS 56, 66, 88
PROTECT-ACS 26
Pseudo-teletype functions 38
PSTACK(X) 68, 103
PSTATUS 86
PTY... 38
PUT 55, 56
QUANTUM(X) 68, 103
question itemvars 6 1
QUICK-CODE 26
RAID 96
ready 67
READYME 70, 103
real constants 89
REALIN 3 7
Reals, range 5
REALSCAN 3 8
RECURSIVE declaration 4
RECURSIVE procedures 7
REDEFINE 47
Reentering programs 94
REF_ITEM 7 6
REFERENCE 7, 8, 18
Reference items 76
RELEASE 33
REMEMBER 29, 30
REMOVE 55, 56
REMOVE, in Foreach 60
RENAME 33
REPLACE-DELIMITERS 46
REQUIRE 9
REQUIRE - indexed by last word of the require statement
REQUIREs, list of 4
RESCHEDULE 73, 103
rescheduling of processes 70
RESERVED 49
Restarting programs 94
RESTORE 29, 30
RESUME 69
RESUME, <options> 69, 103
RESUME, <return item> 69
RETAIN 73, 103
retrieval item expression 65
Retrieval Triple <element> 55, 60
RETURN 24
RETURN statement 17
ROT 24
RPG commands 9 1
RUNME 68, 103
running 67
SAFE declaration 4
SAMEIV 8 5
satisfier group 59
SAY-WHICH 73, 103
SCAN 36
SCHEDULE-ON-CLOCK-INTERRUPTS 79
scheduling of processes 70
scope, of variables 5
SECOND 57, 84
SEGMENT-FILE 10
SEGMENT-NAME 10
SET 53

set booleans 65
Set expressions 65
Set membership <element> 59
SETBREAK 35
SETCP 74, 86
SETFORMAT 40
SETIP 74, 86
Sets, automatic coercion 56
Sets, Derived Sets 65
Sets, initialization 65
Sets, PUT 56
Sets, REMOVE 56
SHORT 21, 98
SIMPLE declaration 4
simple expressions 22
SIMPLE procedures 8
SOURCE-FILE 10, 50
SPROUT 67
SPROUT, <options> 68, 103
START-CODE 26
START-CODE, calling procedures from 28
STDBRK 35
storage reallocation 94
STRING-PDL 9 --
STRING-SPACE 9
String constant, as comment. 1
string constants 89
String descriptors 107
String, declaration 6
STRINGSTACK(X) 68, 103
Substrings 24
SUCCEED 55, 56, 61, 69
SUSPEND 69
suspended 67
SUSPHIM 68, 103
SUSPME 68, 103
Swap statement 14
switches, in command lines 92
symbols, automatic generation of 47
SY STEM-PDL 9
TELLALL 72, 103
TERMINATE 69
terminated 67
THIRD 57, 84
time sharing with processes 79
Triple, Binding Booiean 58
TRIPLES 53
Triples, introduction 51
TRUE, definition 89
TTYIN 38
TTYINL 38
TTYINS 38
type checking, itemvars 53
type conversions, algebraic 21
typed-item-expression 88
TYPEIT 83
unbound 58
UNSTACK-DELIMITERS 46
URSCHD 70
USER1 74
USER2 74
USERCON 42
USERERR 42
USETI 37
USETO 37
VALUE 7, 8, 18
value 58
variables 88
Variables, allocation 9
variables, initialization 9
variables, scope 5
VERSION 10
WAIT 73, 103
wait queue 72
WAITQ 74
WHILE statement 16
WHILEC 49
WORDIN 36
WORDOUT 36
XOR 23