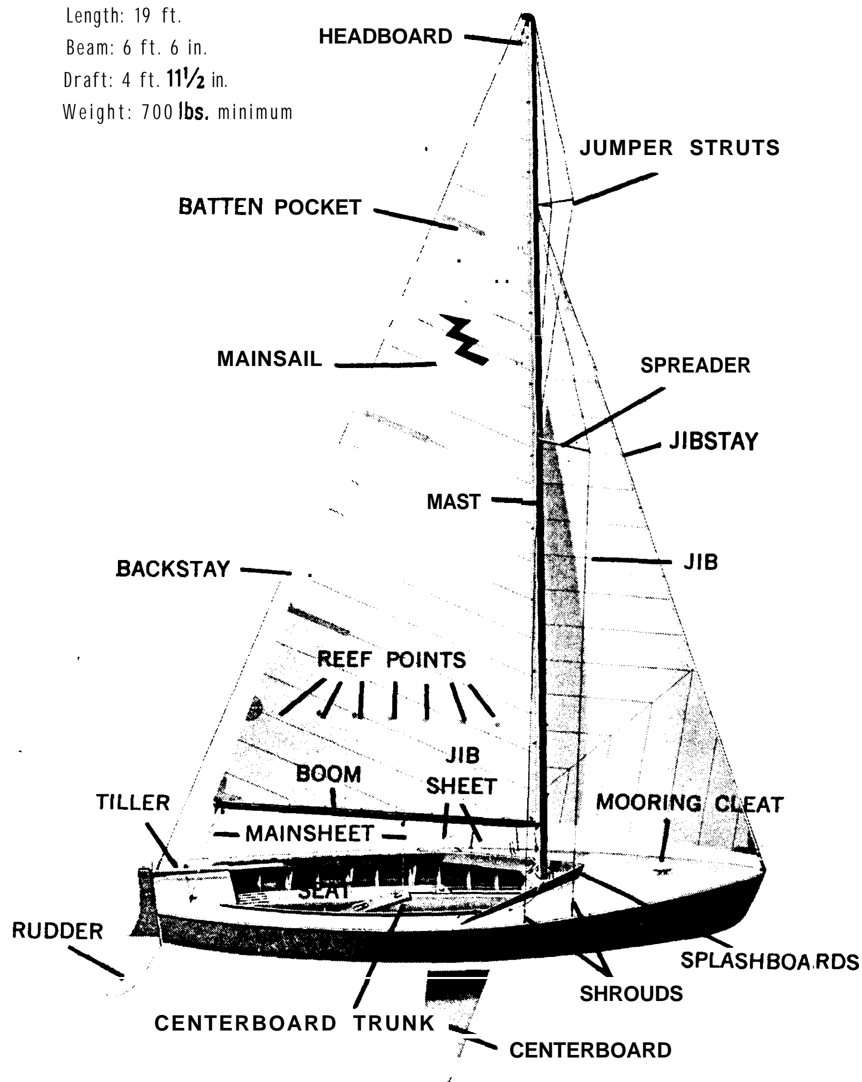
Stanford Artificial Intelligence Laboratory
Memo AIM-290

Computer Science Department
Report No. STAN-B-760575

October 1976

SAIL TUTORIAL



Research sponsored by

National Institutes of Health
and
Advanced Research Projects Agency
ARPA Order No. 2494

COMPUTER SCIENCE DEPARTMENT
Stanford University

Stanford Artificial Intelligence Laboratory
Memo AIM-290

October 1976

Computer Science Department
Report No. STAN-G-76-575

SAIL TUTORIAL

by

Nancy W. Smith
SUMEX-AIM Computer Project
Department of Genetics
Stanford University Medical Center

ABSTRACT

This TUTORIAL is designed for a beginning user of Sail, an ALGOL-like language for the PDP 10. The first part covers the basic statements and expressions of the language; remaining topics include macros, records, conditional compilation, and input/output. Detailed examples of Sail programming are included throughout, and only a minimum of programming background is assumed.

This manual was prepared as part of the SUMEX-AIM computing resource supported by the Biotechnology Resources Program of the National Institutes of Health under grant RR-00735. Printing and preparation for publication were supported by ARPA under Contract MDA903-76-C-0206.

The views and conclusions contained in this document are those of the author(s) and should not be interpreted as necessarily representing the official policies, either expressed or implied, of Stanford University, NIH, ARPA, or the U. S. Government.

Reproduced in the U.S.A. Available from the National Technical Information Service, Springfield, Virginia 22161.

TABLE OF CONTENTS

SECTION PAGE

1 Introduction 1

2 The ALGOL-Part of Sail 2

1 Blocks 2
2 Declarations 2
3 Statements 5
4 Expressions 10
5 Scope of Blocks 13
6 More Control Statements 15
7 Procedures 19

3 Macros 25

4 String Scanning 27

5 Input/Output 30

1, Simple Terminal I/O 30
2 Notes on Terminal I/O for TENEX Sail Only 30
3 Setting Up a Channel for I/O 30
4 Input from a File 37
5 Output to a File 39

6 Records 40

1 Declaring and Creating Records 40
2 Accessing Fields of Records 41
3 Linking Records Together 41

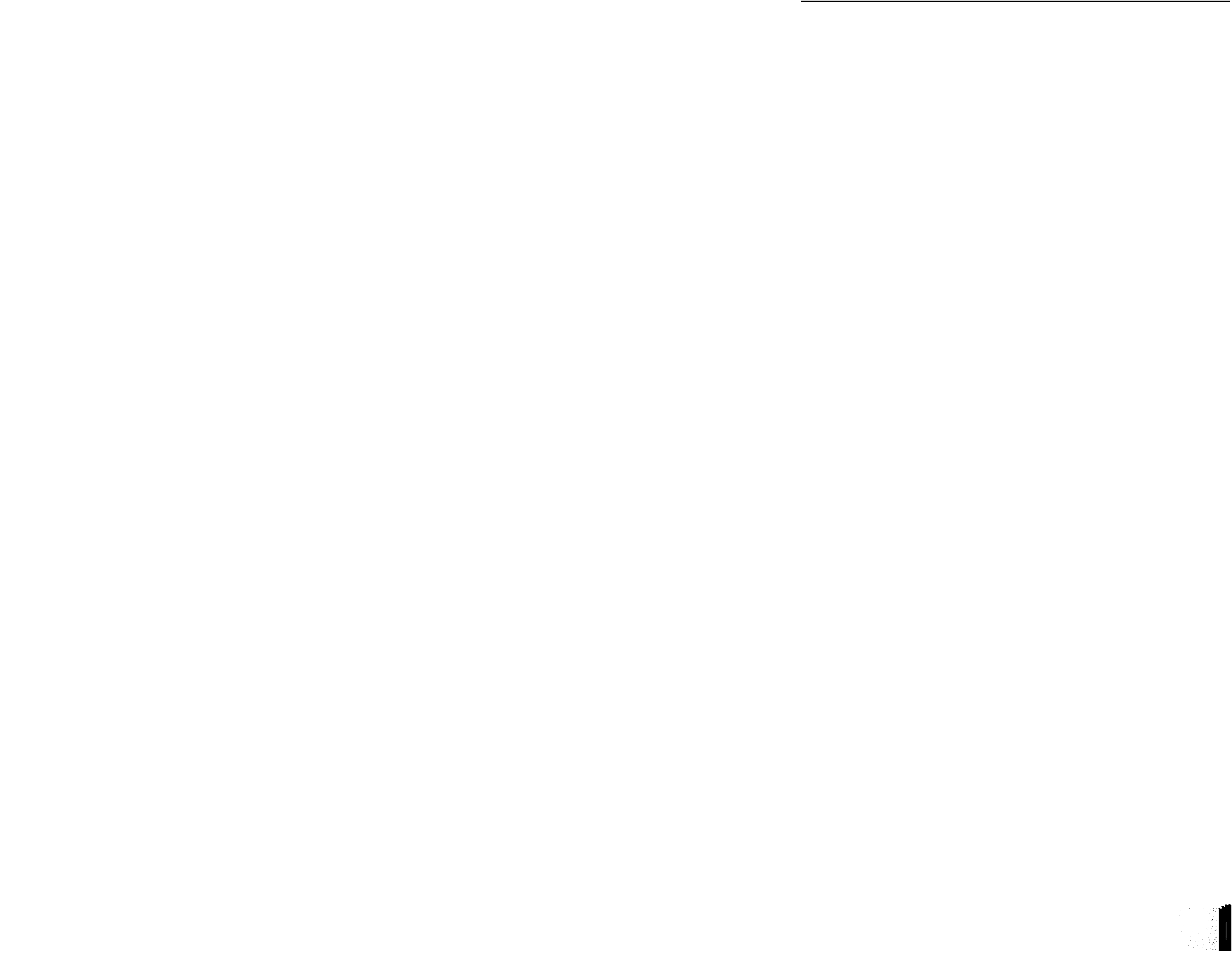
7 Condition al Compilation 44

1 The Load Module 45
2 Source Files 46
3 Macros and Conditional Compilation 47

APPENDIX A: Sail and ALGOL W Comparison 48

REFERENCES 49

INDEX 50



SECTION 1

Introduction

The 'Sail manual [1] is a reference manual containing complete information on Sail but may be difficult for a new user of the language to work with. The purpose of this TUTORIAL * is to introduce new users to the language. It does not deal in depth with advanced features like the LEAP portion of Sail; and uses pointers to the relevant portions of the manual for some descriptions. Following the pointers and reading specific 'portions of the manual will help you to develop some familiarity with the manual. After you have gained some Sail programming experience, it will be worthwhile to browse through the complete reference manual to find a variety of more advanced structures which are not covered in the TUTORIAL but may be useful in your particular programming tasks. The Sail manual also covers use of the BAIL debugger for Sail.

The TUTORIAL is not at an appropriate level for a computer novice. The following assumptions are made about the background of the reader:

1) Some experience with the PDP-10 including knowledge of an editor, understanding of the file system, and familiarity with routine utility programs and system commands. If you are a new user or have previous experience only on a non-timesharing system, you should read the TENEX EXEC MANUAL [7] (for TENEX systems) or the DEC USERS HANDBOOK [6] (for standard TOPS-10 systems) or the MONITOR MANUAL [3] and UUO MANUAL [2] (for Stanford AI Lab users). In addition, you might want to glance through and keep ready for reference: the TENEX JSYS MANUAL [8] and/or the DEC ASSEMBLY LANGUAGE HANDBOOK [5]. Also, each POP-10 system usually has its own introductory materiel for new users describing the operation of the system.

2) Some experience with a programming language--probably FORTRAN, ALGOL or an assembly language. If you have no programming experience, you may need help getting started even with this TUTORIAL. Sail is based on ALGOL so the general concepts and most of the actual statements are the same in what is often called the "ALGOL part" of Sail. The major additions to Sail are its input/output routines. Appendix A contains a list of the differences between the ALGOL W syntax and Sail.

Programs written in standard Sail (which will henceforth be called **TOPS-10** Sail) will usually run on a TENEX system through the emulator (PA1050) which simulates the TOPS-10 UUO's, but such use is quite inefficient. Sail also has a version for TENEX systems which we refer to as TENEX Sail. (The new TOPS-20 system is very similar to TENEX; either TENEX Sail or a new Sail version should be running on TOPS-20 shortly.) Note that the Sail compiler on your system will be called simply Sail but will in fact be either the TENEX Sail or TOPS-10 Sail version of the compiler. Aside from implementation differences which will not be discussed here, the language differences are mainly in the input/output (I/O) routines. And of course the system level commands to compile, load, and run a finished program differ slightly in the TENEX and TOPS-10 systems.

---

* I would like to thank Robert Smith for editing the final version; and Scott Daniels for his contributions to the RECORD section. John Reiser, Les Earnest, Russ Taylor, Marney Beard, and Mike Hinckley all made valuable suggestions.

SECTION 2 .

The ALGOL-Part of Sail .

2.1 Blocks

Sail is a block-ttructurod language. Each block **has** the form:

```
BEGIN

<declarations>
     .

<statements>

     .
END
```

Your **entire** program will be a block with the above format. This program block is a somewhat special block called the **outer** block BEGIN and END **are reserved words** in Sail that mark the beginning **and** end of blocks, with the outermost BEGIN/END pair also marking the beginning and end of your program. (Reserved words are words that automatically mean something to Sail; they are called "reserved" because you should not try to give them your own meaning.)

Declarations are used to give the compiler information about the **data** structures that **you** will be using so that the compiler can set up storage locations of the proper types and associate the desired name with each location.

Statements form the bulk of your program. They **are the actual** commands available in Sail to use for coding **the task at** hand.

. All declarations in each block must precede all statements in that block Here is a very simple one-block program that outputs the square root of 5:

```
                BEGIN
DECLARATIONS ==> INTEGER i;
                REAL x;
STATEMENTS   ==> i ← 5;
                x ← SQRT(i);
                PRINT ("SQUARE ROOT OF ", i,
                      " IS ", x);
                END
```

which will print out on the terminal:

```
SQUARE ROOT OF 5 IS 2.236668 .
```

2.2 Declarations

A list of all the kinds of declarations is given in the Sail manual (Sec. 2.1). In this section we will cover typo **declarations** and array declarations. Procedure declarations will be discussed in Section 2.7. Consult the Sail manual for details on all of the other varieties of declarations listed.

2.2.1 Type Declarat ions

The purpose of type declarations is to tell the compiler what it needs to know to set up the storage locations for your data. There are four **data** types available in the ALGOL portion of Sail:

1) INTEGERs are counting numbers like -1, 0, 1, 2, 3, etc. (Note that commas cannot be used in numbers, e.g., 15724 not 15,724;)

2) REALs are decimal numbers like -1.2, 3.14159, 100087.2, etc.

3) **BOOLEANs** 'are assigned the values **TRUE or FALSE** (which are reserved words). These are predefined for you in Sail (TRUE = -1 and FALSE = 0).

4) STRINGs are a **data** type not found in all programming languages. Very often what you will be working with are not numbers at all but text. Your program may need to output text to the user's terminal while he/she is running the program. It may ask the user questions and input text which is the answer to the question. It may in fact process whole files of text. One simple example of this is a program which works with a file containing a list of words and outputs to a new file the same list of words in alphabetical order. It is possible to do these things in languages with only the integer and real data types but very clumsy. Text has certain properties different from those of numbers. For example, it is very useful

to be able to point to certain of the characters in the text and work with just those temporarily or to take one letter off of the text at a time and process it. Sail has the data type STRING for holding "strings" of **text characters.** And **associated** with the STRING data type are string operations **that** work in **a way** analogous to how the numeric operators (+,-,*, etc.) work with the numeric data types. We write the actual strings enclosed in quotation marks. Any of the characters in the ASCII character set can be used in strings (control characters, letters, numerals, punctuation marks). Some examples of strings are:

```
"OUTPUT FILE= "
"HELP"
"Please type your name."
"aardvark"
"8123456789"
"!""#$%&"
"AaBbCcDdEeFf"

""      (the empty string)
NULL    (also the empty string)
```

Upper and lowercase letters are not equivalent in strings, i.e., "a" is a different string **than "A". (Note that to put a "** in **a** string, you use **""**, e.g., "quote a ""word"""".)

in your programs, you will **have** both **variables** and **constants. We have** already given some examples of constants in **each** of the data types. REAL **and INTEGER constants** are just numbers as you usually see them written (2, 618, -4.35, etc.); the BOOLEAN constants are TRUE **and FALSE; and STRING constants** are a sequence **of** text **characters** enclosed in double quotes (and NULL for the empty string).

Variables are used rather than constants when you know that a value will be needed in the given computation but do not know in advance **what the exact value** will be. For example, you **may** want to add 4 numbers, but the numbers will be specified by the user at **runtime** or taken from **a** data file. Or the numbers may be the results of **previous** computations. You might be computing weekly totals **and** then when you have the results for **each** week adding the four weeks together for **a** monthly **total.** So instead of an expression like 2 + 31 + 25 + 5 you need an expression like **X + Y + Z + W or** WEEK1 + WEEK2 + WEEK3 + WEEK4. This is **done** by declaring (through a declaration) that you will need a variable of a certain data type with a specified name. The compiler will set up **a** storage location of the proper type and enter the name and location in its symbol table. Each time that you have an intermediate result which needs to be stored, you must set up the storage location in advance. When we discuss the various statements available, you will see how values **are** input from the user or from a file or saved from a computation and stored in the appropriate location. The names for these variables are often' referred to as their **identifiers.** identifiers can be as long (or short) as you want. However, if you will be debugging with DDT or using TOPS-10 programs such as the CREF cross-referencing program, you should make your identifiers unique to the first six characters, i.e., DDT can distinguish LONGSYMBOL from LONGNAME but not from LONGSYNONYM because the first 6 characters are the same. Identifiers must begin with a letter but following that can be made up of any sequence of letters and numbers. The characters ! and $ are considered to be letters. Certain **reserved words** and **predeclared identifiers** are unavailable for use as names of your own identifiers. A list of these is given in the Sail manual in Appendices B and C.

Typical declarations are:

```
INTEGER i, j,k;
REAL x,y,z;
STRING s,t;
```

where these are the letters conventionally used as identifiers of the various types. There is no reason why you couldn't have INTEGER x; REAL i; except that other people reading your program might be confused. In some languages the letter used for the variable automatically tells its type. This is not true in Sail. The type of the **variable** is established by the declaration. In general, simple one-letter identifiers like these are used for simple, straightforward and usually temporary purposes such as to count an iteration. (ALGOL W users note that iteration variables must be declared in Sail.)

Most of the variables in your program will be declared and used for a specific purpose and the

name you specify should reflect the use of the variable.

```
INTEGER nextWord, pagolcount;
REAL to ta l, subTota l ;
STRING lastname,firstname;
BOOLEAN partial, abortSwitch, outputsw;
```

Both upper and lowercase fetters are equivalent in identifiers and so the case as well as the use of ! and 8 can contribute to the readability of your programs. Of course, the above examples contain a mixture of styles; you will want to choose some style that looks best to you and use it consistently. The equivalence' of upper and lowercase also means that

```
TOTAL | total | Total | toTal | etc.
```

are all instances of the same identifier. So that while it is desirable to be consistent, forgetting occasionally doesn't hurt anything.

Some programmers use uppercase for the standard words like BEGIN, INTEGER, END, etc. and lowercase for their identifiers. Others reverse this. Another approach is uppercase for actual program code and lowercase for comments. It is important to develop some style which you feel makes your programs as easy to read as possible.

Another important element of program clarity is the format, The Sail compiler is **free** format which means that blank lines, indentations, extra spaces, etc. are ignored. Your whole program could be on one line and the compiler wouldn't know the difference. (Lines should be less than 250 characters if a listing is being made using the compiler listing options.) But programs usually have each statement and declaration on a **separate** line with all lines of each block **indented** the same number of spaces. Some **programmers** put BEGIN and END on lines by themselves and others put them on the closest line of-code. It is very important to format your programs so that **they are easy to read.**

2.2.2 Array Declarations

An array is a data structure designed to let you deal with a group of variables together. For example, if you were accumulating weekly totals over a period of **a** year, it would be cumbersome to declare: .

```
REAL week1, week2,week3,.....,week52;
```

and then have to work with the 52 variables each having a separate name. instead you can declare:

```
REAL ARRAY weeks[1: 521 ;
```

The array declaration consists of one of the data type words (REAL, INTEGER, BOOLEAN, STRING) followed by the word ARRAY followed by the identifier followed by the dimensions of the array enclosed in **[ ]'s.** The dimensions give the bounds of the array. The lower bound does not need to be 1. Another common value for the lower bound is 0, but you may make it anything you like. (The LOADER will have difficulties if the lower bound is **a** number of large positive or negative magnitude.) **You** may declare more than one array in the same declaration provided they are the same type and have the same dimensions. For example, one array might be used for the total employee salary paid in the week which will be a real number, but you might also need to record the total employee hours worked and the **total** profit made (one integer and one real value) so **you** could declare:

```
INTEGER ARRAY hours [1:52];
REAL ARRAY salaries, profits [1:52];
```

These 3 arrays are examples of **parallel arrays.**

it is also possible **to have multi-dimensioned arrays.** A common example is an array used to represent **a** chessboard:

```
INTEGER ARRAY chessboard[1:8,1:8];

1,1  1,2  1,3  1,4  1,5  1,6  1,7  1,8
2,1  2,2  2,3  2,4  2,5  2,6  2,7  2,8
 .    .    .    .    .    .    .    .
 .    .    .    .    .    .    .    .
 .    .    .    .    .    .    .    .
 .    .    .    .    .    .    .    .
 .    .    .    .    .    .    .    .
8,1  8,2  8,3  8,4  8,5  8,6  8,7  8,8
```

In fact even the terminology used is the same. Arrays, like matrices and chessboards, have rows (across) and columns (up-and-down). Arrays which are statically allocated (all outer block and OWN arrays) may have at most 5 dimensions. Arrays which are allocated dynamically may have any number of dimensions.

Each element of the array is a separate variable and can be used anywhere that a simple variable can be used. We refer to the elements by giving the name of the array followed by the particular coordinates (called the **subscripts**) of the given element enclosed in []'s, for example: weeks[34], weeks[27], chessboard [2,5] , and chessboard[8,8].

2.3 Statements

All of the statements available in Sail are listed in the Sail manual (Sec. 1.1 with the syntax for the statements in Sec. 3.1). For now, we will discuss the assignment statement, the PRINT statement, and the IF...THEN statement which will allow us to give some sample programs.

2.3.1 Assignment Statement

Assignment statements are used to assign values to variables:

```
variable ← expression
```

The variable being assigned to and the expression whose value. is being assigned to it are separated by the character which is a backwards arrow in 1965 ASCII (and Stanford ASCII) and is an underbar (underlining character) in 1968 ASCII. The assignment statement is often read as:

```
    variable becomes expression
OR variable is assigned the value of expression
OR variable gets expression
```

You may assign values to any of the four types of variables (INTEGER, REAL, BOOLEAN, STRING) or to the individual variables in arrays.

Essentially, an expression is something that has a value. An expression is not a statement (although we will see later that some of the constructions of the language can be either statements or . expressions depending on the current use). It is most important to remember that an expression can be evaluated. It is a symbol or sequence of symbols that when evaluated produces a value that can be assigned, used in a computation, tested (e.g. for equality with another value), etc. An expression may be

a) a constant

b) a **variable**

c) a construction using constants, variables, and the various operators on them.

Examples of these 3 types of expressions in assignment statements are:

```
DON'T FORGET TO DECLARE VARIABLES FIRST!

    INTEGER i,j;
    REAL    x,y;
    STRING s,t;
    BOOLEAN isw,osw,iosw;
    INTEGER ARRAY arry [1:10];

a)  i ← 2;          COMMENT now i = 2;
    x ← 2.4;        COMMENT now x = 2.4;
    s ← "abc";      COMMENT now EQU(s,"abc");
    isw ← TRUE;     COMMENT now isw = TRUE;
    osw ← FALSE;    COMMENT now osw = FALSE;
    arry[4] ← 22;   COMMENT now arry[4] = 22;

b)  j ← i;          COMMENT now i = j = 2;
    y ← x;          COMMENT now x = y = 2.4;
    t ← s;          COMMENT now EQU(s,"abc")
                           AND EQU(t,"abc");
    arry[8] ← j;    COMMENT i = j = arry[8] = 2;

c)  i ← j + 4;      COMMENT j = 2 AND i = 6;
    x ← 2y - i;     COMMENT y=2.4 AND i=6
                           AND x = -1.2;
    arry[3] ← i/j; COMMENT i=6 AND j=2
                           AND arry[3]=3;
    iosw ← isw OR osw;  COMMENT isw = TRUE
                           AND osw = FALSE
                           AND iosw = TRUE;
```

NOTE1: Most of the operators for strings are different than those for the arithmetic variables. The difference between = and EQU will be covered later.

NOTE2: Logical operators such as AND and OR are also available for boolean expressions.

NOTE3: You **may put** "comments" anywhere in your program by using the word COMMENT followed by the text of your comment and ended with a semi-colon (no semi-colons can appear within the comment). Generally comments are placed between declarations or statements rather than inside of them.

NOTE4: In all our **examples,** you will see that the declarations and statements are **separated** by semi-colons.

In a later section,' we will discuss: 1) type conversion which occurs when the data types of the variable and the expression are not the same, 2) the order of evaluation in the expression, and 3) many more complicated expressions including string expressions (first we need to know more of the string operators).

2.3.2 PRINT Statement

PRINT is a relatively new but very useful statement in Sail. **It** is used for outputting to the user's terminal. You can give it **as** many arguments as you want and the arguments may be of any type. PRINT **first** converts each argument to **a** string if necessary and then outputs it. Remember that only strings can be printed anywhere. Numbers are stored internally as 36-bit words and when they are output in **7-bit** bytes for text the results are very strange. Fortunately PRINT does the conversion to strings for you automatically, e.g., the number 237 is printed **as** the string "237". The format **of** the **PRINT** statement is the word PRINT followed by a list of arguments separated by commas with **the** entire list enclosed in **parent hoses. Each argument may be** any **constant,** variable, or complex expression. For **example,** if you wanted to output the weekly salary totals from a previous example and the **number** of the current week was stored in INTEGER **curWeek,** you might use:

```
PRINT("WEEK ", curWeek,
      ": Salaries ", salaries[curWeek]);
```

which for **curWeek = 28** and the array element **salaries[28] = 27543.82** would print **out:**

```
WEEK 28:Salaries    27543.82
```

NOTE: The printing format for reals (number of leading zeroes printed and places after the decimal point) is discussed in the Sail manual under type conversions.

2.3.3 Built-in Procedures

Using just the assignment statement, the PRINT statement, and three **built-in procedures, we** can write a sample program. Procedures are a very important feature of Sail and you will be writing many of your own. The details of procedure writing and use will be covered in Section 2.7. Without giving any details now, we will just say that some procedures to handle very common tasks have been written for you and are available **as** built-in procedures. The SQRT, **INCHWL** and CVD procedures that we will be using here are all procedures which return values. Examples are:

```
s ← INCHWL;
i ← CVD(s);
x ← 2 + SQRT(i);
```

Procedures may have any number of arguments (or none). **SQRT** and CVS have **a** single argument and **INCHWL** has no arguments (but does return a value). The procedure call is made by writing the procedure **name** followed by the argument(s) in parentheses. **In** the expression in which it is used, the procedure call is equivalent to the **value** that it returns.

SQRT returns the square root of its argument.

CVD returns the result of converting its string' argument to an integer. The string is assumed to contain a number in decimal **representation--** CVO converts strings containing octal numbers, e.g., **after** executing

```
i ← CVD("14724"); j ← CVO("14724");
```

then the following

i = 14724 AND j = 6612

would be true.

INCHWL returns the next line of typing

from the user at the controlling terminal.

NOTE: In TENEX-Sail the **INTTY** procedure is available and SHOULD be used in preference to the **INCHWL** procedure for inputting lines. This may not be mentioned in every example, but is very important for TENEX users to remember.

So, for the statement **s . INCHWL;**, the value of **INCHWL will** be the line typed at the terminal (minus the terminator which **is** usually carriage return). This value is a string and is assigned here to the string variable **s.**

So far we have seen five uses of expressions: as the right-hand-side of the assignment statement, 8s an actual parameter or argument in a procedure call, as an argument to the PRINT statement, for giving the bounds in an array declaration (except for arrays declared in the outer block which must have constant bounds), and for the array subscripts for the elements of arrays. In fact the whole range of kinds of expressions can be used in nearly all the places that constants and variables (which **are particular** kinds of expressions) can be used. Two exceptions to this that we have already seen are **1)** the left-hand-side of the assignment statement (**you** can assign **a** value to a variable **but** not **to a constant** or a more complicated expression) and 2) the array bounds for outer block arrays which **come** at **a** point in the program before any assignments have been made to any of the variables so only constants may be **used--the** declarations in the outer block are before any program statements at all.

In general, any construction that makes sense to you is probably legal in' Sail. By using some of the more complicated **expressions,** , you can save yourself steps in your program. For example,

```
BEGIN
RERL sqroot;
INTEGER numb;
STRING reply;
PRINT("Type number: ");
reply←INCHWL;
numb←CVD(reply);
sqroo t ←SQRT(numb);
PRINT("ANS: ",sqroot);
END;
```

can be shortened by several steps. First, we **can combine** INCHUL with **CVD:**

```
numb ← CW (INCHWL);
```

and eliminate the declaration of the **STRING reply.** Next we can eliminate **numb** and take the SORT directly:

```
[illegible] . SQRT(CVD(INCHWL));
```

At first you might think that we could go a step further to

```
PRINT ("ANS: ",SQRT(CVD(INCHWL)));
```

and we could as far as the Sail syntax is concerned but it would produce **a** bug in our program. We would be printing out **"ANS:"** right after **"Type** number: **"** before the user would have **time** to even **start** typing. But we have considerably simplified our program to:

```
BEGIN
RERL rqroo t;
PRINT ("Typo number:");
sqroot ← SQRT(CVD(INCHWL));
PRINT ("ANS:",sqroot);
END;
```

Remember that intermediate results do not need to be stored unless you will need them again later for something else. By not storing results unnecessarily, you save the extra assignment statement and the storage space by not needing to declare a variable for temporary storage.

2.3.4 IF...THEN Statement

The previous example included no error checking. There are several fundamental programming tasks that cannot be handled with just the assignment and PRINT statements such as 1) conditional tasks like checking the value of a number (is it negative?) and taking action according to the result of the test and 2) looping or iterative tasks so that we could go back to the beginning and ask the user for another number to **be** processed. These sorts of functions are performed by **a** group of statements called control **statements.** In this section we will cover the IF..THEN statement for conditionals. More advanced control statements will be discussed in Section 2.6.

There **are** two kinds of **IF...THEN** statements:

```
IF boolean rxprrssion THEN statement
```

```
IF boolean expression THEN statement
                   ELSE rtatrmnt
```

A boolean expression is an expression whose value is either true or false. A wide variety of expressions can effectively be used in this position, Any arithmetic expression can be a boolean; if its value = 0 then it is FALSE. For any other value, it is TRUE. For now we will just consider the following three cases:

1) BOOLEAN variables (where • rrorsu, base8, and miniVersion are declared as BOOLEANs):

```
IF errorsw THEN
   PRINT("There's been a n error.") ;
IF base8 THEN dig i ts ← "91234567"
          ELSE digits . "0123456789" ;
IF miniVersion THEN counter ← 18
               ELSE counter, ←188;
```

2) Expressions with relational operators such as EQU, =, <, >, LEQ, NEQ, and GEQ:

```
. IF x < currentSmallest THEN
            currentSmallest . x; .
IF divisor NEQ 0 THEN
            quotient←dividend/divisor;
IF i GEQ 0 THEN i←i+1 ELSE i←i-1;
```

3) Complex expressions formed with the logical operators AND, OR, and NOT:

```
IF NOT • rrorsw THEN
   • nsudrrkountrrl ← quot ient;
IF x<0 OR y<0 THEN
   PRINT("Negative numbers not • llotmd.")
   ELSE z ← SQRT(x)+SQRT(y);
```

In the IF..THEN statement, the boolean expression is evaluated. If it is true then the statement following the THEN is executed. If the boolean expression is false and the particular statement has no ELSE part then nothing is done. If the boolean is false and there is an ELSE part then the statement following the ELSE will ba executed.

```
BEGIN BOOLEAN boo l; INTEGER i,j;
bool←TRUE; i←1;   j←1;
IF bool THEN i←i+1;      COMMENT i=2 AND j=1;
IF bool-THEN i←i+1 ELSE j←j+1;
                         COMMENT i=3 AND j=1;
bool←false;
IF bool THEN i←i+1;      COMMENT i=3 AND  j=1;
IF bool THEN i←i+1 ELSE j←j+1;
                         COMMENT i=3 AND j=2;
END;
```

It is VERY IMPORTANT to note that NO semi-colon appears between the statement and the ELSE. Semi-colons are used a) to separate declarations from each other, b) to separate the final declaration from the first statement in the block, c) to separate statements from each other, and d) to mark the end of a comment, The key point to note is that semi-colons are used to separate and NOT to terminate. In some cases it doesn't hurt to put a semi-colon where it is not needed. For example, no semi-colon is needed at the end of the program but it doesn't hurt. However, the format

```
IF • xprossion THEN statement; ELSE statement;
```

makes it difficult for the compiler to understand your code. The first semi-colon marks the end of what could be a legitimate IF...THEN statement and it will be taken as such. Then the compiler is faced with

```
ELSE statement;
```

which is meaningless and will produce an error message.

The following is a part of a sample program which uses several IF...THEN statements:

```
BEGIN BOOLEAN verbosesw; STRING reply;

PRINT("Verbose mode? (Typo Y or N): ");
reply ← INCHWL;    COMMENT    INTTY for TENEX;

IF reply="Y" OR reply="y" THEN verbosesw ← TRUE
ELSE
IF reply="N" OR reply="n" THEN verbosesw←FALSE;

IF verbosesw THEN PRINT("-long msg-")
ELSE PRINT ("-short msg-");

COMMENT now all our messages printrd out to
terminal will be conditional on verbosesw;
END;
```

There are two interesting points to note about this sample program. First is the use of = rather than EQU to check the user's reply. EQU is used to check the equality of variables of type STRING and = is used to check the equality of variables of type INTEGER or REAL. If we were asking the user for a full word answer like "yes" or "no" instead of the single character then we would need the EQU to check what the input string was.

However, in this case where we only have a single character, we can use the **fact that** when a string (either a string variable or a string constant) is put someplace in a program where an integer is expected then Sail automatically converts to the integer which is the ASCII code **for** the **FIRST** character in the string. For example, in the environment

```
STRING str;    str ← "A";
```

all of the following are true:

```
"A"  = str = 65 = '101
"A" NEQ "a"
str NEP "a"
str + 1 = "A" + 1 = '182 = "8"
rtr = "Aardvark"
NOT EQU (str, "Aardvark ")
```

('101 **is an** octal integer constant.)

When you **are** dealing with single character strings (or are only interested in the first character of a string) then you can treat them like integers and use the arithmetic operators like the = operator rather than EQU. In general (over 90% of the time), EQU is slower.

A second point to note in the **above IF...THEN** example **is** the use of **a nested IF...THEN.** The statements following the THEN and the ELSE may be any kind of statement including **another** IF..THEN statement. For example,

```
IF upperOnly THEN letters← "ABC"
   ELSE IF lowerOnly THEN letters ← "abc"
   ELSE lsttsrs ← "ABCabc";
```

This is a very common construction when you **have** a small list of possibilities to check for. **(Note: if there are a large number of cases to be** checked use the CASE statement instead.) The nested **IF..THEN..ELSE** statements save a lot of processing if used properly. For example, without the nesting this would be:

```
IF upperOnly THEN letters ← "ABC";
IF lowerOnly THEN letters ← "abc";
IF NOT upperOnly AND NOT lowerOnly THEN
      letters ← "ABCabc";
```

Regardless of the values **of** upperOnly and lowerOnly, **the** -boolean expreselons in the three **IF..THEN** statements need to be checked. In the nested version, if upperOnly is TRUE then lowerOnly will never be checked. For greatest efficiency, **the most likely case should be the** first one tested in a nested IF...THEN statement. If that likely case is true, no further testing will be done.

To avoid ambiguity in parsing the nested IF..THEN..ELSE construction, the following rule is used: Each ELSE matches up with the last unmatched THEN So that

```
IF ● xpl THEN   IF exp2 THEN s1 ELSE s2;
```

will group the ELSE with the second THEN which is equivalent to

```
IF exp1 THEN
   BEGIN
      IF exp2 THEN s1 ELSE s2;
   END;
```

and also equivalent to

```
IF ● xpl AND ● xp2 THEN s1;
IF ● xpl AND NOT exp2 THEN s2;
```

You can change the structure with BEGIN/END to:

```
IF ● xpl THEN
   BEGIN
      IF ● xp2 THEN s1
   END ELSE s2;
```

which is equivalent to

```
IF sxpl AND exp2 THEN s1;
IF NOT sxpl THEN s2;
```

There is another common use of BEGIN/END in IF-THEN statements. All the examples so far have shown a single simple statement 'to be executed. In fact, you often will have **a** variety of tasks to perform based on the condition tested for. For example, before you make an entry into an array, you may want to check that you are within the array bounds and if so then both make the entry and increment the pointer so **that** it will be ready for the next entry:

```
IF pointer LEQ max THEN
   BEGIN
      data[pointer] ← newEntry;
      pointsrcpointrr + 1;
   END
ELSE PRINT("Array DATA is already full.";;
```

Here we see the use of **a** compound **statement.** Compound statements are exactly, like blocks except that they have no declarations. It would also be perfectly acceptable to use a block with

declarations where the compound statement is used here. In **fact** both blocks and compound statements ARE statements and can be used ANY **place** that **a** simple statement can be used. All of the statements between **BEGIN** and END are executed as a unit (unless **one of the statements** itself . **causes** the flow of execution to be changed).

2.4 Expressions

We have already seen many of the operators used in expressions. Sections 4 and 8 of the Sail manual cover the operators, the' order of evaluation of expressions, and type conversions. Appendix. 1 of the manual gives the word equivalents for the single character operators, e.g., **LEQ** for the less-than-or-equal-to sign, which are not available except at SU-AI. You should read these sections especially for a complete list of the arithmetic and boolean operators available (the **string** operators will be **covered** shortly in this TUTORIAL). A short discussion of type conversion will be given later in #his section but you should also read these sections in **the** Sail manual for complete details on type conversions.

There are three kinds of expressions that we **have** not used yet: assignment, conditional, and case expressions. These **are** much like. the **statements of the same names.**

2.4.1 Assignment Expressions

Anywhere that you can have an expression, you **may** at the same time make an assignment. The value will be used as the value of the expression and also assigned to the given variable. For **example:**

```
IF (reply←INCHWL) = "?" THEN . . . .
COMMENT inputs reply and makes first tort
        on It In single step;

IF (counter←counter+1) > maxEntry THEN . . . .
COMMENT updates counter and chocks it for
        overflow in one step;

counter←ptr←nextloc←0;
COMMENT initializes several variables to 0
       - in one statement;

arry[ptr←ptr+1] ← newEntry ;
COMMENT updates ptr & fills nrxt array
        slot in single stop;
```

Note that the assignment operator has low precedence and so you will often need to use parenthesizing to get **the** proper order of **evaluation.** This **is** an area where many coding errors commonly occur.

```
IF i←j OR boole THEN . . . .
```

is parsed like

```
IF i←(j OR boole) THEN . . . .
```

rather than

```
IF (i←j) OR boole THEN ....
```

See the sections in the Sail manual referenced **above for** a more complete discussion of the order of evaluation in expressions. In general it is the normal order for the arithmetic operators; then the logical operators AND and OR (so that OR has the **lowest** precedence of any operator except the assignment operator); and left to right order is used for two operators at the same level (but the manual gives examples of exceptions). **You can use parentheses** anywhere to specify the order that you want. As an example of the **effect** of left-to-right **evaluation, note that**

```
indexer←2;
arry[indexer] ← (indexer←indexer+1);
```

will put the value **3** in arry[2], since the destination is evaluated before **indexer** is incremented.

A word of caution is needed about assignment **expressions.** Make sure if you put an ordinary assignment in an expression that that expression is in a position where it will ALWAYS be evaluated. Of course,

```
IF i<j THEN i←i+1;
```

will not always increment i but this is the intended result. However, the following is **unintended** and incorrect:

```
IF verbosesw THEN
PRINT("The square root of ",numb," is ",
      sqroot←SQRT(numb)," . ")
ELSE PRINT (sqroot) ;
```

If verbosesw = FALSE, the THEN portion is not executed and the assignment to sqroot is not made. Thus sqroot will not have the appropriate value when it is PRINTed. Assigning the result of a computation to a variable to save recomputing it is an excellent practice but be careful where you put the assignment.

Another very bad place for assignment expressions is following either the AND or CR logical operators. The compiler handles these by performing as little evaluation as possible so in

```
exp1 OR exp2
```

the compiler will first evaluate exp1 and if it is TRUE then the compiler knows that the entire boolean expression is true and doesn't bother to evaluate exp2. Any assignments in exp2 will not be made since exp2 is not evaluated. (Of course, if exp1 is FALSE then exp2 will be evaluated.) Similarly for

```
exp1 AND exp2
```

if exp1 is FALSE then the compiler knows the whole AND-expression is FALSE and doesn't bother evaluating exp2.

As with nested IF...THEN...ELSE statements, it is a good coding practice to choose the order of the expressions carefully to save processing. The most likely expression should be first in an OR expression and the least likely first in an AND expression.

2.4.2 Conditional Expressions

Conditionals can also be used in expressions. These have a more rigid structure than conditional statements. It must be

```
IF booleanexpression THEN exp1 ELSE exp2
```

where the ELSE is not optional.

N. B. The type of a conditional expression is the type of exp1. If exp2 is evaluated, it will be converted to the type of exp1. (At compile time it is not known which will be used so an arbitrary decision is made by always using the type of exp1.) Thus the statement, x←IF flag THEN 2 ELSE y; , will always assign an INTEGER to x. If x and y are REALs then y is converted to INTEGER and then converted to REAL for the assignment to x. X←IF flag THEN 2 ELSE 3.5; will assign either 2.0 or 3.0 to x (assuming x is REAL). Examples are:

```
REAL RRRRY results
        [1: IF miniversion THEN 10 ELSE 1001];

PRINT (IF found THEN words[i]
        ELSE "Word not found.");
COMMENT words[i] must be a string;

profit ← IF (net ← income-cost) > 0 THEN net
        ELSE 0;
```

These conditional expressions will often need to be parenthesized.

2.4.3 CASE Expressions

CASE statements are described in Section 2.6.4 below. CASE expressions are also allowed with the format:

```
CASE integer OF (exp0,exp1,...,expN)
```

where the first case is always 0. This takes the value you give which must be an integer between 0 and N and uses the corresponding expression from the list. A frequent use is for error handling where each, error is assigned a number and the number of the current error is put in a variable. Then a statement like the following can be used to print the proper error message:

```
PRINT(CASE errno OF
        ("Zero division attempted",
         "No negative numbers allowed",
         "Input not a number"));
```

Remember that errno here must range from 0 to 2; otherwise, a case overflow occurs.

2.4.4 String Operators

The STRING, operators are:

```
EQU     Test for string equality:
        s←"ABC"; t←"abc"; test←EQU(s,t);
        RESULT: test = FALSE.
```

8	Concatonatr tuo strings together: s←"abc"; t←"def"; u←s&t; RESULT: EQU(u,"abcdef")= TRUE .
LENGTH	Returns the length of a string: s←"abc"; i←LENGTH(s); RESULT: i = 3 .
LOP	Removes thr first char in a string and returns it: s←"abc"; t←LOP (s) ; RESULT: (EQU(s,"bc") AND EQU(t,"a"))= TRUE .

Although LENGTH and LOP look like procedures **syntactially,** they actually compile code "in-line". This means that they compile very fast code. However,. one unfortunate side-effect is that LOP cannot be used as a statement, i.e., you cannot say LOP(s); if you just want to throw away the first character of the string. You must always either 'use or assign the character returned by LOP even if you don't want it for anything, e.g., junk←LOP (s); . Another point to note about LOP is that it actually removes the character from the original string. If you will need the intact string again, you should make **a** copy of it before **you** start **LOP'ing,** e.g., tempCopy←s; .

A little background on the implementation of **strings** should help you to use them more efficiently. Inefficient use of strings can be a significant inefficiency in your programs. **Sail** sets up **an** area of memory called string **space** where all the actual strings are stored. **The runtime** system increases the size of this area dynamically as it begins to become full. The **runtime** system also performs garbage, collections to retrieve space taken by strings that are no longer needed so that the space can be reused. The text of the strings is stored in string space. Nothing is put in string space until you actually **specify what the** string is to be, i.e., by an assignment statement. At the time of the **declaration,** nothing is put in string **space. Instead** *the compiler sets up a 2-word string **descriptor** for each string declared. The first word contains in its left-half an indication of whether the string is a constant or a variable and in its right-half the length of the string. The second word is a byte pointer to the location of the start of the string in **string** space. At the time of the declaration, the length will be zero and the byte pointer word will be empty since the string is not yet in string space.

From this we can see that LENGTH and LOP are very efficient operations. LENGTH picks up the length from **the** descriptor word; and LOP decrements the length by 1, picks up the character designated by the byte pointer, and increments the byte pointer. LOP does not need to do anything with string space. Concatenations with & are however fairly inefficient since in general new strings must be created. For s & t, there is usually no way to change the descriptor words to come up with the pew string (unless s and t are already adjacent in string space). Instead both s and t must be copied into a new string in string space. In general since the pointer is kept to the beginning of the string, it is less expensive to look at the beginning than the end. On the other hand, when concatenating, it is better to keep building onto the end of a given string rather than the beginning. The **runtime** routines know what is **at** the end of string space and, if you happen to concatenate to the end of the last string put in, the routines can do that efficiently without needing to copy the last string.

Assigning one string variable to another, e.g., for making a temporary copy of the string, is also fast since the string descriptor rather than the text is copied.

These are general guidelines rather than strict rules. Different programs will have different specific needs and features.

2.4.5 Subst rings

Sail provides a way of dealing with selected subportions of strings called substrings. There are two different ways to designate the desired substring:

```
s[i TO j]
s[i FOR j]
```

where [i TO j] means the substring starting at the ith character in the string through the jth character and [i FOR j] is the substring starting at the ith character that is j characters long. The · numbering starts with 1 at the first character on the left. The special symbol INF can be used to refer to the last character (the rightmost) in the string. So, s[INF FOR 1] is the last character; and s[7 TO INF] is all but the first six characters. If you are using a substring of a

string array element then the format is arry[index][i TO j].

Suppose you have made the assignment s←"abcdef". Then,

```
s[1 TO 3]              is "abc"
s[2 FOR 3]             is "bcd"
s[1 TO INF]            is "abcdef"
s[INF-1 TO INF]        is "ef"
s[1 TO 3]&"X"&s[4 TO INF]  is "abcXdef" .
```

Since substrings are parts of the text of their source strings, it is a very cheap operation to break a string down, but is fairly expensive to build up a new string out of substrings.

2.4.6 Type Conversions

If you use **an** expression of one type where another type **was** expected, then automatic type conversion is performed. For example,

```
INTEGER I;
I ← SQRT(5);
```

will cause 5' to be converted to real (because SQRT expects **a** real argument) and the square root of 5.0 to be automatically converted to an integer before it is assigned to i which was declared as an integer variable and can only have integer values. As noted in Section 4.2 of the Sail manual, this conversion is done by truncating the real **value.**

Another example of automatic type conversion that we **have used** here in many of the sample programs is:

```
IF reply="Y" THEN . . . . .
```

where the = operator always expects integer or real arguments rather than strings. Both the value of the string variable reply and the string constant "Y" will be converted to integer values before the equality test. The manual shows that this conversion, string-to-integer, is performed by taking the first character of the string and using its ASCII value. Similarly converting from integer to string is done by interpreting the integer (or just **the** rightmost seven bits if it is less than 0 or-it is too large--that is any number over 127 or '177) as an ASCII code and using the **character that the code represents as the string. So, for example,**

```
STRING s;
s ← '181 & '182 & '183;
```

will make the string "ABC".

The other common conversions that we have seen are integer/real to boolean and string to boolean. Integers and reals are true if non-zero; strings are true if they have a non-zero length and the first character of the string is not the NUL character (which is ASCII code 0).

You may **also** call one of the built-in type conversion procedures explicitly. **We have used CVD** extensively to convert strings containing digits to the integer number which the digits represent. CVD and a number of other useful type conversion procedures are described in Section 8.1 of the Sail manual. Also this section discusses the SETFORMAT procedure which is used for' specifying the number of leading zeroes and the maximum length of the decimal portion of the real when printing. SETFORMAT is extremely useful if you will be outputting numbers as **tables** and need to have them automatically line up vertically.

2.5 Scope of Blocks

So far we have seen basically only one use of inner blocks. With the IF..THEN statement, we saw that you sometimes need a block rather than a simple statement following the THEN or ELSE so that a group of statements can be executed **as** a unit.

In **fact,** blocks can be used within the program any place that you can use a single statement. Syntactically, blocks are statements. A typical program might look like this:

```
BEGIN "prog"
  .
  .
  BEGIN "initialization"
    .
    .
  END "initialization"

  BEGIN "main part"

    BEGIN "process data"
      .
      .
      BEGIN "output results"
        .
```

```
            END "output results"

         END "process data"

      END "main part"

      BEGIN "finish up"

      .
      END "finish up"

   END "prog"
```

The declarations in each block establish variables which can only be used in the given block. So another reason for using inner blocks is to manage variables needed for a specific short range task.

Each block can (should) have a block name. The name is given in quotes following the BEGIN and END of the block. The case of the letters, number of spaces, etc. are important (as in string constants) so that the names "MAIN LOOP", "Main Loop", "main loop", and "Main loop" are all different and will not match. There are several advantages to using block names: your programs are easier to read, the names will be used by the debugger and thus will make debugging easier, and- the compiler will check block names and report any mismatches to help you pinpoint missing END's (a very common programming error).

The above example shows us how blocks may nest. Any block which is completely within the scope of another block is said to be nested in that block. In any, program, all of the inner blocks are nested in the outer block. Here, in addition to all the blocks being within the "prog" block, we find "output results" nested in "process data" and both "output results" and "process data" nested in "main part". The three blocks called "initialization", "main part" and "finish up" are not nested with relation to each other but are said to be at the same **level.** None of the variables declared in any of these three blocks is available to any of the others. In order to have a variable shared by these blocks, we need to declare it in a block 'which is "outer" to all of them, which is in this case the very outermost block **"prog".**

Variables are available in the **block in** which they are declared and in all the blocks nested in that block UNLESS the inner block also has a variable of the same name declared (a very bad idea in general). The portion of the program, i.e., the blocks, in which the variable is available is called the **scope of the variable.**

```
BEGIN "main"
INTEGER i,j;
i←5;
j←2;
PRINT("CASE A: i=",i,"   j=",j);
  BEGIN "Inner"
    INTEGER i,k;
    i←10;
    k←3;
    PRINT("CASE  B:  i=",i," j=",j,"   k=",k);
    j←4;
  END "inner";
PRINT("CASE C: i=", i,"   j=",j);
END "main"
```

Here we cannot access k except in block "inner". The variable j is the same throughout the entire program. There are 2 variables both named i. So the program will print out:

```
CASE    A: i=5 j=2
CASE    B: i=10  j=2  k=3
CASE    C:  i=5 j=4
```

Variables are referred to as local variables in the block in which they are declared. They are called global variables in relation to any of the blocks nested in the block of their declaration. With both a local and a global variable of the same name, the local variable takes precedence. There are three relationships that a variable can have to a block:

1) It is inaccessible to the block if the variable is declared in a block at the same level as the given block or it is declared in a block nested within the given block.

2) It is local to the block if it is declared in the block.

3) It is global to the block if it is declared in one of the blocks that the given block is nested within.

Often the term "global variables" is used specifically to mean the variables declared in the outer block which are global to all the other blocks.

In reading the Sail manual, you will see the terms: allocation, deallocation, initialization, and **reinitialization.** It is not important to completely understand the implementation details, but it is extremely important to understand the effects. The key point is that allocating storage for data can be handled in one of two ways. Storago allocation refers to the actual setting up of data locations in memory. This can be done 1) at compile time or 2) at **runtime.** If it is done at **runtime** then we say that the allocation is dynamic. Basically, it is arrays which are dynamically allocated (excluding outer block arrays and other arrays which are declared **as** OWN). LISTS, SETS, and RECORDS which we have not discussed in this section are also. allocated dynamically. The following are allocated **at** compile time and are NOT dynamic: scalar variables (INTEGER, BOOLEAN, REAL and STRING) except where the scalar variable is in **a** recursive procedure, outer block arrays, and other OWN arrays. ALGOL users should note this **as an** important ALGOL/Sail difference.

Dynamic storage (inner block arrays, etc.) will be allocated **at** the point that the block is entered and deallocated when the block is exited. This makes for quite efficient use of large amounts of storage space that serve a short term need. **Also,** it allows you to set variable size bounds for these arrays since **the value does not need** to be known at compile time.

At the time that storage is allocated, it is also initialized. This means **that** the initial value is **assigned---NULL for strings** and 0 for integers, reals, and booleans. Since arrays are, allocated **each** time the block is entered, they are reinitialiied each time. We have not yet seen any cases where the same block is executed more than once but this is very frequent with the iterative and looping control statements.

Scalar variables and outer block arrays are not dynamically allocated. They are allocated by the compiler and will receive the inital null or zero value when the program is loaded but they will never be reinitialized. While you are not in the block, the variables are not accessible to you **but they** are not deallocated so they will have the same value when you enter the block the next time as when-you exited it on the previous **use.** Usually you will find that this is not what you want. You should initialize all local scalar **variables** yourself somewhere near the start of the block--usually to NULL for strings and 0 for arithmetic variables unless you need some other specific initial value. You should also initialize all global **scalars** (and outer block arrays) at. the start of your program to be on the safe side. They are initialized for you when the compiled program is later run, but their values will not be reinitialized if the program is restarted while already in core and the results will be very strange.

One exception is the blocks in RECURSIVE PROCEDUREs which do have all non-OWN variables properly handled and initialized as recursive calls are made on the blocks.

If you should want to clear an array, the command

```
ARRCLR (arry)
```

will clear arry (set string arrays to NULL and arithmetic to 0). For arithmetic (NOT string) arrays,

```
RRRCLR (arry, val)
```

will set the elements of arry to **val.**

See Sections 2.2-2.4 of the Sail manual for more information on OWN, SAFE, **and PRELOADED arrays and Section 8.5 for the ARRBLT and ARRTRAN routines for moving the contents of** arrays.

2.6 More Control Statements

2.6.1 FOR Statement

The FOR statement is used 'for a definite number of iterations. **Many** times you will want to repeat certain code a specific number of times (where usually the number in the sequence of repetitions is also important in the code performed). For example,

```
FOR i ← 1 STEP 1 UNTIL 5 DO
    PRINT(i, " ", SQRT(i));
```

which will print out a **table of the square roots** of the numbers 1 to 5.

The syntax of the (simple) FOR statement is

```
FOR variable←starting-value STEP increment
    UNTIL end-value DO statement
```

The iteration **variable** is assigned the **starting**-value and tested to check if it exceeds the **end**-value; if it is within the range then the statement after the DO is executed (otherwise the FOR statement is finished). This completes the first execution of the FOR-loop.

Next the increment is added to the variable and it is tested to see if it now exceeds the **end**-value. If it does then the statement is not executed again and the FOR statement is finished. If it is within the maximum '(or equal to it) then the statement is executed again but all instances of the iteration variable in the statement will now have the new value. This incrementing and checking and executing loop is repeated until the iteration variable exceeds the end-value.

For those users familar with GOT0 statements and LABELs, the following two program **fragments for computing** r n s . **FACT(n)** a r e equivalent.

```
ans ← 1;
FOR i ← 2 STEP 1 UNTIL n DO  •  nscanse il
```

is equivalent to:

```
          ant ← 1;
          ж • 2;
loop:     IF i> n THEN GOT0 bryond ;
          ant   ← ans * i;
          I ← I + 1;
          GOTO loop;
beyond :
```

There is considerable dispute on whether or not the use of GOT0 statements should be encouraged and if so under what conditions. These statements are available in Sail but will not be discussed in this Tutorial.

Very often FOR-loops are used for indexing through arrays. For example,, if you are computing averages, you **will** need to add together numbers which might be stored in an array. The following program allows a teacher to input the total number of tests taken and a list of the scores; then the program returns the average score.

```
BEGIN "averager"
REAL average; INTEGER numbTests, total ;
● verrgm-numbTostsetotale8;
COMMENT remember t o initialize variables;
PRINT("Total number of torts: ");
numbTests←CVD(INCHWL);

  BEGIN "useArray"
   INTEGER ARRAY testScores[1:numbTests];
   COMMENT array has variable bounds so must
         be i n inner block;
   INTEGER i ;
   COMMENT f o r use a s the iteration variable;

   FOR I ← 1 STEP 1 UNTIL numbTests DO
       BEGIN "fillarray"
           PRINT("Test Score #",i," : ");
           testScores[i] ← CVD(INCHWL);
       END "fillarray";

   FOR i ← 1 STEP 1 UNTIL numbTes ts DO
       total←total+testScores[i];
   COMMENT notr that total was initialized to
           0 above;

   END "useArray";

 IF numbTests neq 0 THEN average←total/numbTests;
 PRINT("The average i s ",average,".");
 END "averager";
```

In the first FOR-loop, we see that i is used in the PRINT statement to tell the user which test score is wanted then it is used again as the array 'subscript to put the score into the i'th element of the array. Similarly it is used in the second FOR-loop to add the i'th element to the cumulative total.

The iteration variable, start-value, increment, end end-value can all be reals as well as integers. They can also be negatives (in which case the maximum is taken as a minimum). See the Sail manual for details on other variations where multiple values can be given for more complex statements (these aren't used often). One point to note is that in Sail the end-value expression is evaluated each time through the loop, while the increment value is evaluated only at the beginning if it is a complex expression, as opposed to a constant or a simple variable. This means that for efficiency, if your loop will be performed very many times you should not have very complicated expressions in the end-value position. If you need to compute the end-value, do it before the FOR-loop and assign the value to a variable that can be used in the FOR-loop to save having to recompute the value each time. This doesn't save much and probably isn't worth it for 5 or 10 iterations but for 500 or 1000 it can be quite a savings. For example use:

```
max←(ptr-offset)/2;
FOR i←offset STEP 1 UNTIL max DO s;
```

rather than

```
FOR i←offset STEP 1 UNTIL (ptr-offset)/2 DO s;
```

2.6.2 WHILE...DO Statement and DO...UNTIL Statement

Often you will want to repeat code but not know in **advance** how **many** times. Instead the iteration will be finished when a certain condition is met. This is called **indefinite iteration** and is done with either a **WHILE...DO** or a **DO...UNTIL** statement.

The syntax of WHILE statements is:

```
WHILE boolean-expression DO statement
```

The boolean is checked and if FALSE nothing is done. If TRUE the statement is executed and then the boolean is checked again, etc.

For example, suppose we want to check through the elements of an integer array until we find an element containing a given number n:

```
INTEGER ARRAY arry [1:max];
ptr ← 1;
WHILE (arry[ptr] NEQ n) AND (ptr < max) DO
    ptr←ptr+1;
```

If the array element currently pointed to by ptr is the number we **are** looking for OR if the ptr is at the upper bound of the array then the **WHILE** statement is finished. Otherwise the **ptr** is incremented and the boolean (now using the next element) is checked again. When the **WHILE...DO** statement is finished, either ptr will point to the **array** element with the number or **ptr=max** will mean that nothing was found.

The **WHILE...DO** statement is equivalent to the following format with LABELs and the **GOTO** statement:

```
loop:   IF NOT boolean expression THEN
                GOTO beyond;
        statement;
        GOTO loop;
beyond:
```

The DO...UNTIL statement is very similar except **that 1) the statement is always executed the first time and then the check is made before each subsequent loop through and 2) the loop** continues UNTIL the boolean becomes true rather than WHILE it is true.

```
DO statement UNTIL boolean-expression
```

For example, suppose we want to get a series of names from the user and store the names in **a** string array. We will finish inputting the names when **the user types a bare carriage-return** (which results in a string of length 0 from **INCHWL** or **INTTY).**

```
i←0;
DO PRINT("Name #",i←i+1," is: ")
        UNTIL   (LENGTH (names[i] ← INCHWL) = 0 );
```

The equivalent of the **DO...UNTIL** statement using LABELs and the **GOTO** statement is:

```
loop:   statement;
        IF NOT boolean expression THEN GOTO loop;
```

Note that the checks in the WHILE...DO and **DO...UNTIL** statements **are** the reverse of each other. WHILE...DO continues as long as the expression is true but **DO...UNTIL** continues as long as the expression **is** NOT true. So that

```
WHILE i < 100 DO .....
```

is equivalent to

```
DO ..... UNTIL i GEQ 100
```

except that the statement is guaranteed to be executed at least once with the **DO...UNTIL** but not with the **WHILE...DO.**

The WHILE and DO statements can be used, for **example, to** check **that a** string which we **have** input from the user is really an integer. CVD stops converting if it hits **a** non-digit and returns the **results** of the conversion to that point but does not give an error indication so that **a** check of this sort should probably be done on numbers input from the user before CVD is **called.**

```
INTEGER numb, char;
STRING reply,temp; BOOLEAN error;
PRINT("Type the number: ");
DO
  BEGIN
    error+FALSE;
    temp+reply+INCHWL;
    UHILE LENGTH (temp) DO
      IF NOT ("0" LEQ (char+LOP(temp)) LEO "9")
        THEN error+TRUE;
    IF error THEN PRINT("Oops, try again: ");
  END
  UNTIL NOT error;
numb+CVD(reply);
```

2.6.3 DONE and CONTINUE Statements

Even with definite and indefinite iterations available, there will still be times when you need a greater degree of control over the loop. This is accomplished by the DONE and CONTINUE statements which can be used in any loop which begins with DO, e.g.,

```
FOR i+1 STEP 1 UNTIL j DO . . .
DO . . . UNTIL rxp
WHILE  xp DO . . .
```

(See the manual for a discussion of the NEXT statement which is not often used.) DONE means to abort execution of the entire FOR, DO...UNTIL or WHILE...DO statement immediately. CONTINUE means to stop executing the current pass through the loop and continue to the next iteration.

Suppose a string array is being used as a "dictionary" to hold a list of 100 words and we want to look up one of the words which is now stored in a string called **target**:

```
FOR i + 1 STEP 1 UNTIL 188 DO
    IF EQU(words[i],target) THEN DONE;
IF i>100 THEN PRINT(target," not found.");
```

If the target is found, the FOR-loop will stop regardless of the current value of i. Note that the iteration variable can be checked after the loop is terminated' to determine whether the DONE forced the termination (i LEQ 100) or the target was never found and the loop terminated naturally (i > 100).

If the loops are nested then the DONE or CONTINUE applies to the innermost loop unless there are names on the blocks to be executed by each loop and the name is given explicitly, e.g., OONE "someloop". With the DONE and CONTINUE statements, we can now give the complete code to be used for the sample program given earlier where a number was accepted from the user and the square root of the number was returned. A variety of error checks are made and the user can continue giving numbers until finished. In this example, block names will be used with DONE and CONTINUE only where they are necessary for the correctness of the program; but use of block names everywhere is a good practice for clear programming.

```
BEGIN "prog"     STRING temp,reply; INTEGER numb;,

UHILE TRUE DO
COMMENT a very common construction which just
      loops until DONE;
  BEGIN "processnumb"
    PRINT("Type a number, <CR> to end, or ?:");
    UHILE TRUE DO
      BEGIN' "checker"
        IF NOT LENGTH(temp+reply+INCHWL) THEN
          DONE "processnumb";
        IF reply = "?" THEN
          BEGIN
            PRINT("..helptext & reprompt..");
            CONTINUE j
            COMMENT defaults to "checker";
          END;
        UHILE LENGTH (temp) DO
          IF NOT ("0" LED LOP (temp) LEQ "9") THEN
            BEGIN
              PRINT("Oops, try again: ");
              CONTINUE "checker";
            END;
        IF (numb+CVD (rep l y)) < 0 THEN
          BEGIN
            PRINT("Negat ivo, try again: ");
            CONTINUE j
          ENO;
        DONE;
        COMMENT if all the chocks have boon
              passed thrn done;
      END "checker";
    PRINT("The Square Root of ",numb," is ",
          SQRT(numb),".")j
    COMMENT now we go back to top of loop
          for next inputj
  END "processnumb";
END "prog"
```

2.6.4 CASE Statement

The CASE statement is similar to the CASE expression where S0,S1,...Sn represent the statements to be given at these positions.

```
CASE integer OF
     BEGIN
     S0;
     ;   COMMENT the empty statement;
     S2;
     .
     .
     Sn
     END;
```

where ;'s are included for those cases where no action is to be taken. Another version of the CASE statement is:

```
CASE integer OF
     BEGIN
     [0] S0;
     [4] S4; COMMENT cases can be skipped;
     [3] S3; COMMENT need not be in order;
     [5] S5;
   [6] [7] S6; COMMENT may be same statement;
     [8]   S8;

     .
     [n] Sn
     END;
```

where explicit numbers in []'s are given for the cases to be included.

It is very IMPORTANT not to use a semi-colon after the final statement before the END. Also, do NOT use CASE statements if you have a sparse number of cases spread over a wide range because the compiler will make a giant table, e.g.,

```
CASE number OF
     BEGIN
     [0] S0;
     [1000] S1000;
     [2000] S2000
     END;
```

would produce a 2001 word table!

Remember that the first case is 0 not 1. An example is using a CASE statement to process lettered options:

```
INTEGER char;
PRINT("Type A,B,C,D, or E :");
char←INCHWL;
CASE char-"A" OF
COMMENT "A"-"A" is 0, and is thus case 0;
     BEGIN
     <code for A option>;
     <code for B option>;
     .
     .
     <code for E option>
     END;
```

2.7 Procedures

We have been using built-in procedures and in fact would be lost without them if we had to do all our own coding for the arithmetic functions, the interactions with the system like Input/Output, and the general utility routines that simplify our programming. Similarly, good programmers would be lost without the ability to write their own procedures. It takes a little time and practice getting into the habit of looking at programming tasks with an eye to spotting potential procedure components in the task, but it is well worth the effort.

Often in programming, the same steps must be repeated in different places in the program. Another way of looking at it is to say that the same task must be performed in more than one context. The wry this is usually handled is to write a **procedure** which is the sequence of statements that will perform the task. This procedure itself appears in the declaration portion of one of the blocks in your program and we will discuss later the details of how you declare the procedure. Essentially at the time that you are writing the statement portion of your program, you can think of your procedures as black boxes. You recognize that you have an instance of the task that you have designed one of your procedures to perform and you include at that point in your sequence of statements a **procedure** call statement. The procedure will be invoked and will handle the task for you. In the simplest case, the procedure call is accomplished by just writing the procedure's name.

For example, suppose you have a calculator-type program that accepts an arithmetic expression from the user and evaluates it. At suitable places in the program you will have checks to make sure that no divisions by zero are being attempted. You might write a procedure called zeroDiv which prints out a message to the user saying that a zero division has occurred, repeats

the current arithmetic expression, and asks if the user would like to see the prepared help text for the program. Every time you check for zero division anyplace in your program 'and find it, you will call, this procedure with the statement:

```
zeroDiv;
```

and it will do everything it is supposed to do.

Sometimes the general format of the task **will** be the same but some details will be .**different.** These **cases** can be covered by writing a **parameterized procedure.** Suppose that we wanted something like our **zeroDiv** procedure, but more general, that would handle a number of other kinds of errors. It still needs to print out a description of the error, the current expression being evaluated, and a suggestion that the user consult the help **text;** but the description of the error will be different depending on what the error was. We accomplish this by using a variable when we write the procedure; in this case an integer variable for the error number. The procedure includes code to print out the appropriate message for each error number; and the integer variable ● □□■□ is added to the parameter **list** of the procedure. Each of the par-meters is a variable that will need to have a value associated with it automatically at the time the procedure is called. (Actually arrays and other procedures can also be parameters; but they will be discussed later.) We won't worry about the handling of parameters in procedure declarations now. We are concerned with the way the parameters are specified in the procedure call. **Our** procedure ● rrorHand **ler** will have one integer parameter so we call it with the expression to be associated with the integer variable ● rrno given in parentheses following the procedure name in the procedure call. For example,

```
● rrorHandler(0)
● rrorHandler(1)
errorHandler(2)
```

would be the valid calls possible if we had three different possible errors.

If there is-more than one parameter, they are put in the order given in the declaration and separated -by -commas. **(Arguments** is another term used for the actual parameters supplied in a procedure call.) Any expression can be used for the parameter, e.g., for the built-in procedure **SQRT:**

```
SQRT(4)
SQRT (numb)
SQRT (CVD(INCHWL))
SQRT(numb/divisor)
```

When Sail compiles the code for these procedure calls, it first includes code to associate the appropriate values in the procedure call with the variables given in the parameter list of the procedure declaration and then includes the code to execute the procedure. When ● rrorHand ier PRIM% the error message, the variable ● □□■□ 'will have the appropriate value associated with it. This is not an assignment such **as** those done by the assignment statement and we will also be discussing calls by REFERENCE as well **as** calls by VALUE; but we don't need to go into the details of the actual implementation -- see the manual if you are interested in how procedure calls are implemented and arguments pushed on the stack.

Just '**as** we often perform the same task many times in a given program so there are tas'ks performed frequently in many programs by many programmers. The authors of Sail have written procedures for **a** number of such tasks which can be used by everyone. These are the built-in procedures (CVD, **INCHWL,** etc.) and are actually declared in the Sail **runtime** package so all that is needed for **you** to use them is placing the procedure calls **at** the appropriate places. Thus these procedures are indeed black boxes when they are used.

However, for our own procedures, we do need to write the code ourselves. An example of a useful procedure is one which converts **a** string argument to all uppercase characters. First, the program with the procedure call to **upper at the** appropriate place and the position marked where the procedure declaration will go:

```
BEGIN
STRING reply,name;
***procedure dociaration here***

PRINT("Type RERD, WRITE, or SEARCH: ");
reply←upper(INCHWL);
IF EQU(reply, "READ") THEN . . . .
    ELSE IF EQU(reply,"WRITE") THEN . . . .
    ELSE IF EQU(reply,"SEARCH") THEN . . . .
    ELSE ....;
END;
```

We put the code for the procedure right in the

procedure declaration which goes in the declaration portion of any block. Remember that the procedure must be declared in a block which will make it accessible to the blocks where you are going to use it; in the same way that a variable must be declared in the appropriate place. Also, any variables that appear in the code of the procedure must already be declared (even in the declaration immediately preceding the procedure declaration is fine).

Here is the procedure declaration for upper which should be inserted at the marked position in the above code:

```
STRING PROCEDURE upper (STRING rawstring);
    BEGIN "upper"
    STRING tmp; INTEGER char;
    tmp←NULL;
    WHILE LENGTH (rawstring) DO
        BEGIN
        char←LOP(rawstring);
        tmp←tmp&(IF "a" LEQ char LEQ "z"
                THEN char-'40 ELSE char);
        END;
    RETURN (tmp);
    END "upper";
```

The syntax is:

```
typo-quaiifior PROCEDURE identifier ;
    statement
```

for procedures with no parameters OR

```
typo-quaiiflor PROCEDURE identifier
    ( parrmotor-list ); statement
```

where the parameter-list is enclosed in ()'s and a semi-colon **precedes** the statement (which is often called the **procedure** body). The **<type-qualifier>'s** will be discussed shortly.

The parameter list includes the names and types of the parameters and must NOT have a semi-colon following the final item on the list. Examples are: .

```
PROCEDURE offerHelp;
INTEGER PROCEDURE findWord
    (STRING target; STRING ARRRY words);
SIMPLE PROCEDURE ● rrorH&ndlor
    (INTEGER ● rrno);
RECURSIVE INTEGER PROCEOURE factorial
    (INTEGER number);
PROCEDURE sortEntries
    (INTEGER ptr, f irst; REAL ARRRY unsorted);
STRING PROCEDURE upprr (STRING rawString);
```

Each of these now needs a procedure body.

```
PROCEDURE offerHelp;

BEGIN "offerHelp"
COMMENT the procedure name is usually used
    as block name;
PRINT("Would you like help (Y or N):");
IF upper(INCHWL) = "Y" THEN PRINT("..help..")
    ELSE RETURN;
PRINT("Would you like more hrip (Y or N):");
IF upper(INCHWL) = "Y" THEN
    PRINT("..more help..");
END "offerHelp";
```

This offers a brief help text and if it is rejected then RETURNs from the procedure without printing anything. A RETURN statement may be included in any procedure at any time. Otherwise the brief help message is printed and the extended help offered. After the extended help message is printed (or not printed), the procedure finishes and returns without needing a specific RETURN statement because the code for the procedure is over. Note that we can use procedure calls to other procedures such as upprr provided that we declare them in the proper order with upper declared before offerHelp.

PROCEDURE declarations will usually have **type-qualifiers**. There are two kinds: 1) the simple types--INTEGER, STRING, BOOLEAN, and REAL and 2) the special ones--FORWARD, RECURSIVE, and SIMPLE.

FORWARD is typically used if two procedures call each other. This creates a problem because a procedure must be declared before it can be called. For example, if offerHelp called upper, and upper also called offerHelp then we would need:

```
FORURRD STRING PROCEDURE upper
    (STRING rawstring);

PROCEDURE offerHelp;
  BEGIN "offerHelp"
  . . .
  <code for offerHelp including call to upper>
  . . .
  END "offerHelp";

STRING PROCEDURE upper (STRING rawstring);
```

```
BEGIN "upper"
   . . .
<code for upper including call to offerHelp>
   . . .
END "upper" ;
```

The FORWARD declaration does not include the body but does include the parameter list (if any). This declaration gives the compiler enough information about the upper procedure for it to process the offerHelp procedure. FORWARD is also used when there is no order of declaration of a series of procedures such that every procedure is declared before it is used. FORWARD declarations can sometimes be eliminated by putting one of the procedures in the body of the other, which can be done if you don't need to use both of them later.

RECURSIVE is used to qualify the declaration of any procedure which calls itself. The compiler will add special handling of variables so that the values of the variables in the block are preserved when the block is called again and restored after the return from the recursive call. For example,

```
RECURSIVE INTEGER PROCEDURE factorial
    (INTEGER i);
RETURN(IF i=0 THEN 1 ELSE factorial(i-1)*i);
```

The compiler adds some overhead to procedures that can be omitted if you do not use any complicated structures. Declaring procedures SIMPLE inhibits the addition of this overhead. However, there are severe restrictions on SIMPLE procedures; and also, BAIL can be used more effectively with non-SIMPLE procedures. So the appropriate use of SIMPLE is during the optimization stage (if any) after the program is debugged. At this time the SIMPLE qualifier can be added to the short, simple procedures which will save some overhead. The restrictions on SIMPLE procedures are:

1) Cannot allocate storage dynamically, i.e., no non-OWN arrays can be declared in SIMPLE procedures.

2) Cannot do GO TO's outside of themselves (the GO TO statement has not been covered here).

3) Cannot, if declared inside other procedures, make any use of the parameters of the other procedures.

Procedures which are declared as one of the simple types (REAL, INTEGER, BOOLEAN, or STRING) are called typed procedures as opposed to untyped procedures (note that the SIMPLE, FORWARD, and RECURSIVE qualifiers have no effect on this distinction). Typed procedures can return values. Thus typed procedures are like FORTRAN functions and untyped procedures are like FORTRAN subroutines. The type of the value returned corresponds to the type of the procedure declaration. Only a single value may be returned by any procedure. The format is RETURN(expression) where the expression is enclosed in ()'s. Procedure upper which was given above is a typed procedure which returns as its value the uppercase version of the string. Another example is:

```
REAL PROCEDURE averager
    (INTEGER ARRAY scores; INTEGER max);
BEGIN "averager"   REAL total; INTEGER i;
total ← 0;
FOR i ← 1 STEP 1 UNTIL max DO
    total ← total + scores[i];
IF max NEQ 0 THEN RETURN(total/max)
    ELSE RETURN (0);
END "averager";
```

We might have a variety of calls to this procedure:

```
testAverage ← averager (testScores,numberScores);
salaryAverage ← averager(salaries,numberEmployees);
speedAverage ← averager(speeds,numberTrials);
```

where testScores, salaries, and speeds are all INTEGER ARRAYs.

Procedure calls can always be used as statements, e.g.,

```
1) IF divisor=0 THEN errorHandler(1);
2) offerHelp;
3) upper(text);
```

but as in 3) it makes little sense to use a procedure that returns a value as a statement since the value is lost. Thus typed procedures which return values can also be used as expressions, e.g.,

```
reply←upper(INCHWL);
PRINT(upper(name));
```

It is not necessary to have a RETURN statement

in untyped procedures. If you do have a RETURN statement in an untyped procedure it CANNOT specify a value; and if you have a RETURN statement in a typed procedure it MUST specify a value to be returned. If there is no RETURN statement in'a typed procedure then the value returned will be garbage for integer and real procedures or the null string for string procedures; this is not good coding practice.

Procedures frequently will **RETURN(true)** or **RETURN(false)** to indicate success or a problem. For example, a procedure which is supposed to get a filename from the user and open the file will return true if successful and false if no file was actually opened:

```
IF getFile THEN processInput
          ELSE errorHandler(22) ;
```

This is quite typical code where you can see that all the tasks have been procedurized. Many programs will have 25 pages of procedure declarations and then only 1 or 2 pages of actual statements calling the appropriate procedures at the appropriate times. In fact, programs can be written with pages of procedures and then only a single statement to call the main procedure.

Basically there are two ways of giving information to a procedure and three ways of returning information. To give information you can 1) use parameters to pass the information explicitly or 2) make sure that the appropriate values are in global variables at the time of the call and code the procedures so that they access those varirbles. There are several disadvantages to the latter approach although it certainly does have its uses.

First, once a piece of information has been assigned to a parameter, the coding proceeds smoothly. When you write the procedure call, you can check the parameter list and see at a glance what arguments you need. If you instead use a global variable then you need to remember to make sure it has the right value at the time of each procedure call. In fact in a complicated program you will have enough trouble remembering the name of the variable. This is one of the beauties of procedures. You can think about the task and all the components of the task end code them once and. then when you are in the middle of another larger task, you only need to give the procedure name and the values for all the parameters (which are clearly specified in the parameter list so you don't have to remember them) and the subtask is taken care of. If you don't modularize your programs in this way, you are juggling too many open tasks at the same time. Another approach is to tackle the major tasks first and every time you see a subtask put in a procedure call with reasonable arguments and then later actually write the procedures for the subtasks. Usually a mixture of these approaches is appropriate; and you will also find yourself carrying particularly good utility procedures over from one program to another, building a library of your own general utility routines.

The second advantage of parameters over global variables is that the global variables will actually be changed by any code within the procedures but variables used as parameters to procedures will not. The changing of global variables is sometimes called a **side-effect** of the procedure.

Here are a pair of procedures that illustrate both these points:

```
BOOLEAN PROCEDURE Ques1 (STRING s);
BEGIN "Ques1"
IF "?"=LOP(s) THEN RETURN (true)
              ELSE RETURN(false);
END "Ques1";

STRING str;
BOOLEAN PROCEOURE Ques2;
BEGIN "Ques2"
IF "?"=LOP(str) THEN RETURN (true)
                ELSE RETURN (false);
END "Ques2";
```

The second procedure has these problems: 1) we have to make sure our string is in the string variable str before the procedure call and 2) str is actually modified by the LOP so we have to make sure we have another copy of it. With the first procedure, the string to be checked can be anywhere and no copy is needed. For example, if we want to check a string called command, we give Ques1(command) and the LOP done on the string in Ques1 will not affect command.

Informrtion can be returned from procedures in three ways:

1) With a RETURN(value) statement.

2) Through global variables. You may sometimes actually want to change a

global variable. Also, procedures can only return a single value so if you have several values being generated in the procedure, you may use global variables for the others.

3) Through REFERENCE parameters. Parameters can be either VALUE or REFERENCE. By default all scalar parameters are VALUE and array parameters are REFERENCE. Array parameters CANNOT be value; but scalars can be declared as reference parameters. Value parameters as we have seen are simply used to pass a value to the variable which appears in the procedure. Reference parameters actually associate the variable address given in the procedure call with the variable in the procedure so that any changes made will be made to the calling variable.

```
PROCEDURE manyReturns
(REFERENCE INTEGER i, j,k,l,m);
    BEGIN
      i←i+1; j←j+1; k←k+1; l←l+1; m←m+1;
    ENO;
```

when called with

```
manyReturns(var1,var2,var3,var4,var5);
```

will actually change the var1,..,var5 variables themselves. Arrays are always called by reference. This is useful; for example, you might have a

```
PROCEDURE sorter (STRING ARRAY arry);
```

which sorts a string array alphabetically. It will actually do the sorting on the array that you give it so that the array will be sorted when the procedure returns. Note that arrays cannot be returned with the RETURN statement so this eliminates the need for making all your arrays global as a means of returning them.

See the Sail manual (Sec. 2) for details on using procedures as parameters to other procedures.

SECTION 3

Macros

Sail macros are basically string substitutions made in your source code by. the **scanner** during compilation. Think of your source file as being read by a scanner that substitutes definitions into the token stream going to **a** logical "inner compiler". Anything that one can do with **macros,** one could have done without them by editing the file differently. Macros **are' used for** several **purposes.**

They are used to define named constants, e.g.,

```
BEGIN
REQUIRE "{}{}" DELIHITERS;
DEFINE maxSize = {100};
REAL ARRAY arry [1:maxSize];
```

The {}'s are used as **delimiters** placed around the right-hand-side of the macro definition. Wherever the token MXSize appears, the scanner wilt substitute **100** before the code is compiled. These substitutions of the source text on the right-hand-side of the DEFINE for the token on the left-hand-side wherever it subsequently **'appears in the source file** is called **expanding the** macro. The above array declaration after macro expansion is:

```
BEG IN
REAL ARRAY arry [1:100];
```

which is more efficient than using:

```
BEGIN INTEGER maxSize;
maxSize←100;
    BEGIN
    REAL ARRAY arry [1:maxSize];
```

Also, in this **example,** the use of the integer variable **for** assignment of the **maxSize** means **that the** array-bounds declaration is variable rather than constant **so** it must be in an inner' block; with the macro, **maxSize** is a constant so the array can be declared anywhere.

Other advantages to using macros to define names for constants are **1)** a name like **maxSize** used in your code is easier to understand than an arbitrary number when you or someone else is reading through the program and 2) **maxSize** will undoubtedly appear in many contexts in the program but if it needs to be changed, e.g., to 200, only the single definition needs changing. If you had used 100 instead of **maxSize** throughout the program then you would have to change each 100 to **200.**

Before giving your DEFINEs **you** should require some delimiters. **{}{}, [][],** or **<><>** are good choices. If **you** don't require any delimiters then the defaults are **""""** which are probably a poor choice since they make it hard to define string constants. The first pair of delimiters given in the REQUIRE statement are for the **right-hand-**side of the DEFINE. See the Sail manual for details on use of the second pair of delimiters.

DEFINEs may appear anywhere in your program. They **are** neither statements nor declarations. REQUIREs can be either declarations or statements so they can also go anywhere in your program.

Another use of macros is to define octal characters. If you have tried to use any of the sample programs here **you** will have discovered a glaring bug. Each time we have output our results with the PRIM: statement, no account has been taken of the need for a CRLF (carriage return and line feed) sequence. So all the lines will run together. Here are 4 possible solutions to the problem:

```
    1) PRINT("Some text.", ('15&'12));

    2) PRINT("Some text.
");

    3) STRING crlf;
       crlf←"
";     PRINT ("Some text.",crlf);

    4) REQUIRE "{}" DELIHITERS;
       DEFINE crlf={"
"};    PRINT("Some text.",crlf);
```

The first solution is hard to type frequently with the **octals.** (In general, concatenations should be avoided if possible since new strings must usually be created for them; but in this case with only constants in the concatenation, it will be done **at** compile time so that is not **a** consideration.) The second solution with the

string extending to the next line to get the **crlf** is unwieldy to use in your code. The fourth solution is both the easiest to type and the most efficient.

You may also **want** to define a number of the other commonly used control characters:

```
REQUIRE "<><>" DELIMITERS;
DEFINE ff = <('14&NULL)>,
       lf = <('12&NULL)>,
       cr = <('15&NULL)>,
      tab = <('11&NULL)>,
     ctl0 = <'17>;
```

The characters which will be used as arguments in the PRINT statement must be forced to be strings. If ff = <'14> were used; then **PRINT(ff)** would print the number 12 (which is '14) rather than to print a formfeed **because** PRINT would treat the '14 as an integer. For all the other **places** that you can use these single character definitions, they will work correctly whether defined as strings or integers, e.g.,

```
IF char = ctl0 THEN . . . .
```

as well as

```
IF char = ff THEN . . . .
```

Note that string constants like **'15&'12** and **'14&NULL** do not ordinarily need **parenthesizing but ('15&'12)** and **('14&NULL)** were used above. **This** is a little trick to compile more efficient code: The compiler will not ordinarily recognize these **as** string constants when they appear in the middle of a concatenated string, e.g.,

```
"....line1..."&'15&'12&"....line2..."
```

but with the proper parenthesizing

```
"....line1..."&('15&'12)&"....line2..."
```

the compiler will treat the **crlf** as a string constant at compile time and not need to do a concatenation on '15 and '12 every time at **runtime.**

Another very common use of macros is to **"personalize"** the Sail language slightly. Usually macros of this sort are used either to save repetitive-typing of long sequences or to make the code where they **are used** clearer. (Be careful--this can be carried overboard.)

Here are some sample definitions followed by an example of their use on the next line:

```
REQUIRE "<><>" DELIMITERS;

DEFINE upto = <STEP 1 UNTIL>;
    FOR i upto 10 DO ....;

DEFINE ! = <COMMENT>;
    i←i+1;              ! increment i here;

DEFINE forever = <WHILE TRUE>;
    forever DO ....;

DEFINE eif = <ELSE IF>;
    IF ... THEN ....
      EIF .... THEN ....
      EIF .... THEN ....;
```

Macros may also have parameters:

```
DEFINE append(x,y) = <x←x&y>;
    IF LENGTH(s) THEN append(t,LOP(s));

DEFINE inc(n) = <(n←n+1)>,
       dec(n) = <(n←n-1)>;
    IF inc(ptr) < maxSize THEN .....
COMMENT watch that you don't forget
       needed parentheses here;

DEFINE ctrl(n) = <("n"-'100)>;
    IF char = ctrl(O) THEN abortPrint;
```

As we saw in some of the sample macros, the macro does not need to be **a** complete statement, expression, etc. It can be just a fragment. Whether or not you want to use macros like this is **a** matter of personal taste. However, it is quite clear that something like the following' is simply terrible code although **syntactically** correct (and rumored to have actually occurred in **a** program):

```
DEFINE printer = <PRINT(>;
    printer "Hi there.");
```

which expands to

```
PRINT("Hi there.");
```

On the other hand, those who completely shun macros **are** erring in the other direction. One of the best coding practices in Sail is to DEFINE all constant parameters such **as** array bounds.

SECTION 4

String Scanning

We have not yet covered Input/Output which is one of the most important topics. Before we do that, however, we will cover the SCAN function for reading strings. SCAN which reads existing strings is very similar to INPUT which is used to read in text from a file.

Both SCAN and INPUT use break tables. When you are reading, you could of course read the entire file in at once but this is not what you usually want even if the file would all fit (and with the case of SCAN for strings it would be pointless). A break table is used to 1) set up a list of characters which when read will terminate the scan, 2) set up characters which are to be omitted from the resulting string, and 3) give instructions for what to do with the break character that terminated the scan (append it to the result string, throw it away, leave it at the new beginning of the old string, etc.). During the course of a program, you will want to scan strings in different ways, for example: scan and break on a non-digit to check that the string contains only digits, scan and break on linefeed (lf) so that you get one line of text at a time, scan and omit all spaces so that you have a compact string, etc. For each of these purposes (which will have different break characters, omit characters, disposition of the break character, and setting of certain other modes available), you will need a different break table. You are allowed to set up as many as 54 different break tables in a program. These are set up with a SETBREAK command.

A break table is referred to by its number (1 to 54). The GETBREAK procedure is used to get the number of the next free table and the number is stored in an integer variable. GETBREAK is a relatively new feature. Previously, programmers had to keep track of the free numbers themselves. GETBREAK is highly recommended especially if you will be interfacing your program with another program which is also -assigning table numbers and may use the same number for a different table. GETBREAK will know about all the table numbers in use. You assign this number to a break table by giving it as the first argument to the SETBREAK function. You can also use RELBREAK(table#) to release a table number for reassignment when you no longer need that break table.

```
SETBREAK(table#, "break-characters",
        "omit-charactors", "modes") ;
```

where the first argument is an integer and the ""s around the other arguments here are a standard way of indicating, in a sample procedure call, that the argument expected is a string. For example:

```
REQUIRE "<><>" DELIHITERS;
DEFINE if = <'12>, cr = <'15>,  ff = <'14>;
INTEGER lineBr,nonDigitBr,noSpaces;

SETBREAK(lineBr←GETBREAK, if, ff&cr, "ins"))
SETBREAK(noSpaces←GETBREAK, NULL, "","ina");
SETBREAK(nonDigitBr←GETBREAK, "0123456789",
        NULL, "xns");
```

The characters in the "break-characters" string will be used as the break characters to terminate the SCAN or INPUT. SCAN and INPUT return that portion of the initial string up to the first occurrence of one of the break-characters.

The characters in the "omit-characters" string will be omitted from the string returned.

The "modes" establish what is to be done with the break character that terminated the SCAN or INPUT. Any combination of the following modes can be given by putting the mode letters together in a string constant:

CHARACTERS USED FOR BREAK CHARACTERS:

"I" (inclusion) The characters in the break-characters string are the set of characters which will terminate the SCAN or INPUT.

"X" (eXclusion) Any character except those in the break-characters string will terminate the SCAN or INPUT, e.g., to break on any digit use:

```
INTEGER tbi;
SETBREAK(tbl←GETBREAK,"0123456789",NULL,"i");
```

and to break on any non-digit use:

```
INTEGER tbl;
SETBREAK(tbl←GETBREAK,"0123456789","","x");
```

where NULL or "" can be used to indicate no characters are being given for that argument.

DISPOSITION OF BREAK CHARACTER:

"S" (skip) The character which actually terminates the SCAN or INPUT will be "skipped" and thus will not appear in the result string returned nor will it be still in the original string.

"A" (append) The terminating character will be appended to the end of the result string.

"R" (retain) The terminating character will be retained in its position in the original string so that it will be the first character read by the next SCAN or INPUT.

OTHER MISCELLANEOUS MODES:

"K" This mode will convert characters to be put in the result string to uppercase.

"N" This mode will discard SOS line numbers if any and should probably be used for break tables which will be scanning text from a file. This is a very good Sail coding practice even if it seems highly unlikely that an SOS file will ever be given to your' program.

```
"result-string" ← SCAN(@"source", table#, @brchar);
```

In these sample formats, the ""s mean the argument is a string and the @ prefix means that the argument is an argument by reference.

When you call the SCAN function, you give it as arguments 1) the source string, 2) the break table number and 3) the name of an INTEGER variable where it will put a copy of the character that terminated the scan. Both the source string and the break character integer are reference parameters to the SCAN procedure and will have new values when the procedure is finished. The following example illustrates the use of the SCAN procedure and also shows how the "S", "A", and "R" modes affect the resulting strings with the disposition of the break character.

```
INTEGER skipBr, appendBr, retainBr, brchar;
STRING result, skipStr, appendStr, retainStr;

SETBREAK(skipBr←GETBREAK,"*",NULL,"s");
SETBREAK(appendBr←GETBREAK,"*",NULL,"a");
SETBREAK(retainBr←GETBREAK,"*",NULL,"r");

skipStr←appendStr←retainStr←"first*second";
result ← SCAN(skipStr, skipBr, brchar);
COMMENT EQU(result,"first") AND
        EQU(skipStr,"second");

result ← SCAN(appendStr, appendBr, brchar);
COMMENT EQU(result,"first*") AND
        EQU(appendStr,"second");

result ← SCAN(retainStr, retainBr, brchar);
COMMENT EQU(result,"first") AND
        EQU(retainStr,"*second");

COMMENT in each case above brchar = "*"
        after the SCAN;
```

Now we can look again at the break tables given above:

```
SETBREAK(lineBr,lf,ff&cr,"ins");
```

This break table will return a single line up to the lf. Any carriage returns or formfeeds (usually used as page marks) will be omitted and the break character is also omitted (skipped) so that just the text of the line will be returned in the result string. The more conventional way to read line by line where the line terminators are preserved is

```
SETBREAK(readLine,lf,NULL,"ina");
```

Note here that it is extremely important that lf rather than cr be used as the break character since it follows the cr in the actual text. Otherwise, you'll end up with strings like

```
text of line<cr>
<lf>text of line<cr>
<lf>
```

instead of

```
trxt of line<cr><lf>
text of line<cr><lf>
```

After the SCAN, the brchar variable can be either the break character that terminated the scan (lf in this case) or 0 if no break character was encountered and the scan terminated by reaching the end of the source string.

```
DO processLine(SCAN(str,readLine,brchar))
  UNTIL NOT brchar;
```

This code would be used if you had a long multi-lined text stored in a string and wanted to process it one line at a time with PROCEDURE processLine.

```
SETBREAK(nonDigitBr,"0123456789",NULL,"xs");
```

This break table could be used to check if a number input from the user contains only digits.

```
WHILE true 0 0
    BEGIN
    PRINT("Type a number: ");
    reply←INCHWL;          ! INTTY for TENEX;
    SCAN(reply,nonDigitBr,brchar);
    IF brchar THEN
      PRINT(brchar&NULL," is not a digit.",crlf)
      ELSE OONE;
    END;
```

Here the value of brchar (converted to a string constant since the integer character code will probably be meaningless to the user) was printed out to show the user the offending character. There are many other uses of the brchar variable particularly if a number of characters are specified in the break-characters string of the break table and different actions are to be taken depending on which one actually was encountered.

```
SETBREAK(noSpaces,NULL," ","ina");
```

Here there are no break-characters but the omit-character(s) will be taken care of by the scan, e.g.,

```
str←"a b c d";
result←SCAN(str,noSpaces,brchar) ;
```

will return "abcd" as the result string.

If you need to scan a number which is stored in a string, two special scanning functions, INTSCAN and REALSCAN, have been set up which do not require break tables but have the appropriate code built in:

```
integerVar ← INTSCAN("number-string",@brchar);
realVar . REALSCAN("number-string",@brchar);
```

where the integer or real number read is returned; and the string argument after the call contains the remainder of the string with the number removed. We could use INTSCAN to check if a string input from a user is really a proper number.

```
PRINT("Type the number: ");
reply ← INCHWL;   ! INTTY for, TENEX;
numb ← INTSCAN(reply,brchar);
IF brchar THEN error;
```

SECTION 5

Input /Out put

5.1 Simple Terminal I/O

We have been doing input/output (I/O) from the controlling terminal with INCHWL (or INTTY for TENEX) and PRINT. A number of other Teletype I/O routines are listed in the Sail manual in Sections 7.5 and 12.4 but they are less often used. Also any of the file I/O routines which will be covered next can be used with the TTY: specified in place of a file. Before we cover file f/O, a few comments are needed on the usual terminal input and output.

The INCHWL (INTTY) that we have used is like an INPUT 'with the source of input prespecified as the terminal and the break characters given as the line terminators. Should you ever want to look at the break character which terminated an INCHWL or INTTY, it will be in a special variable called !SKIP! which the Sail runtimes use for a wide variety of purposes. INTTY will input a maximum of 200 characters. If the INTTY was terminated for reaching the maximum limit then !SKIP! will be set to -1. Since this variable is declared in the runtime package rather than in your program, if you are going to be looking at it, you will need to declare it also, but as an EXTERNAL, to tell the compiler that you want the runtime variable.

```
EXTERNAL INTEGER !SKIP!;
PRINT("Number followed by <CR> or <ALT>: ");
reply←INCHWL;        ! INTTY for TENEX;
IF !SKIP!=cr THEN ......
    ELSE IF !SKIP!=alt THEN .....
```

Altmode (escape, enter, etc.) is one of the characters which is different in the different character sets. The standard for most of the world including both TOPS-10 and TENEX is to have altmode as '33. At some point in the past TOPS-10 used '176. This is now obsolete; however, the SU-AI character set follows this convention but does so incorrectly. It uses '175 as altmode. This will present a problem for programs transported among sites. It also partially explains why most systems when they believe they are dealing with a MODEL-33 Teletype or other uppercase only terminal (or are in @RAISE mode in TENEX) will convert the characters '173 to '176 to altmodes.

5.2 Notes on Terminal I/O for TENEX Sail Only

If you are programming in TENEX Sail, you should use INTTY in preference to the various teletype routines listed in the manual. TENEX does not have a line editor built in. You can get the effect of a line editor by using INTTY which allows the user to edit his/her typing with the usual ↑A, ↑R, ↑X, etc. up until the point where the line terminator is typed. If you use INCHWL, the editing characters are only DEL to rubout one character and ↑U to start over. Efforts have been made in TENEX Sail to provide line-editing where needed in the various I/O routines when accessing the controlling terminal. Complete details are contained in Section 12 of the Sail manual.

TENEX also has a non-standard use of the character set which can occasionally cause problems. The original design of TENEX called for replacing crlf sequences with the '37 character (eol). This has since been largely abandoned and most TENEX programs will not output text with eol's but rather use the standard crlf. Eol's are still used by the TENEX system itself. The Sail input routines INPUT, INTTY, etc. convert eol's to crlf sequences. See the Sail manual for details, if necessary; but in general, the only time that you should ever have a problem is if you input from the terminal with some routine that inputs a single character at a time, e.g., CHARIN. In these cases you will need to remember that end-of-line will be signalled by an eol rather than a cr. The user of course types a cr but TENEX converts to eol; and the Sail single character input functions do not reconvert. to cr as the other Sail input functions do.

5.3 Setting Up a Channel for I/O

Now we need I/O for files. The input and output operations to files are much like what we have done for the terminal. CPRINT will write arguments to a file as PRINT writes them to the terminal. It is also possible with the SETPRINT

command to specify that you would rather send your PRINT's to a file (or to the terminal AND a named file). See the manual for details.

There are a number of other functions available for I/O in addition to INPUT and CPRINT, but they all have one common feature that we have not seen before. Each requires as first argument a channel number. The CPU performs I/O through input/output channels. Any device (TTY:, LPT:, DTA:, DSK:, etc.) can be at the other end of the channel. Note that by opening the controlling terminal (TTY:) on a channel, you can use any of the input/output routines available. In the case of directory devices such as DSK: and DTA:, a filename is also necessary to set up the I/O. There are several steps in the process of establishing the source/destination of I/O on a numbered channel and getting it ready for the actual transfer. This is the area in which TOPS-10 and TENEX Sail have the most differences due to the differences in the two operating systems. Therefore separate sections will be included here for TOPS-10 and TENEX Sail and you should read only the one relevant for you.

5.3.1 TOPS- 10 Sail Channel and File Handling

Routines for opening and closing files in TOPS-10 Sail correspond closely to the UUO's available in the TOPS-10 system. The main routines are:

```
GETCHAN OPEN LOOKUP ENTER RELEASE
```

Additional routines (not discussed here) are:

```
USETI USETO MTAPE CLOSE CLOSIN CLOSO
```

5.3.1.1 Device Opening

```
chan← GETCHAN;
```

GETCHAN obtains the number of a free channel. On a TOPS-10 system, channel numbers are 0 through '17. GETCHAN finds the number of' a channel not currently in use by Sail and returns that number. The. user is advised to use GETCHAN to obtain a channel number rather than using absolute channel numbers.

```
OPEN(chan, "device", mode, inbufs,
    outbufs, mount, Ubrchar, @eof);
```

The OPEN procedure corresponds to the TOPS-10 OPEN (or INIT) UUO. OPEN has eight parameters. Some of these refer to parameters that the OPEN UUO will need; other parameters specify the number of buffers desired, with other UUO's called by OPEN to set up this buffering; still other parameters are internal Sail bookkeeping parameters. .

The parameters to OPEN are:

1) CHANNEL: channel number, typically the number returned by GETCHAN.

2) "DEVICE": a string argument that is the name of the device that is desired, such as "DSK" for the disk or "TTY" for the controlling terminal.

3) MODE: a number indicating the mode of data transfer. Reasonable values are: 0 for characters and strings and '14 for words and arrays of words. Mode '17 for dump mode transfers of arrays is sometimes used but is not discussed here.

4) INBUFS: the number of input buffers that are to be set up.

5) OUTBUFS: the number of output buffers.

6) COUNT: a reference parameter specifying the maximum number of characters for the INPUT function.

7) BRCHAR: a reference parameter in which the character on which INPUT broke will be saved.

8) EOF: a reference parameter which is set to TRUE when the file is at the end. ,

The CHANNEL, "DEVICE", and MODE parameters are passed to the OPEN UUO; INBUFS and OUTBUFS tell the Sail runtime system how many buffers should be set up for data transfers; and the COUNT, BRCHAR and EOF variables are cells that are used by Sail bookkeeping. N.B.: many of the above parameters have additional meanings as given in the Sail manual. The examples in this section are intended to demonstrate how to do simple things.

```
RELEASE(chan);
```

The RELEASE function, which takes the channel number as an argument, finishes all the input and output and makes the channel available for other use.

The following routine illustrates how to open a device (in this case, the device is only the teletype) and output to that device. The CPRINT function, which is like PRINT except that its output goes to an arbitrary channel destination, is used.

```
BEG IN
INTEGER OUTCHAN;

OPEN(OUTCHAN ← GETCHAN,"TTY",0,0,2,0,0,0);
COMMENT
        (1) Obtain a channoi number, using
GETCHAN, and save it in variable OUTCHAN.
        (2) Specify device TTY, in mode 8,
uith 0 input and 2 output buffrrs.
        (3)Ignore the COUNT, BRCHAR, and EOF
variables, which are typically not needed if
the file is only for output. ;

CPR INT (OUTCHAN, "Message for OUTCHRN
");
COMMENT Actual data transfor.;

RELEASE(OUTCHAN);
COMMENT Close channoi;
END;
```

The following example illustrates how to read text from a device, again using the teletype as the device.

```
BEGIN
INTEGER INCHAN, INBRCHRR, INEOF;

OPEN(INCHAN← GETCHAN, "TTY", 0, 2, 8, 288,
    INBRCHAR, INEOF );
COMMENT
  Opens the TTY in mode 0 (characters), uith
  2 input buffers, 0 output buffers. At most
  200 charactors will be road in uith each
  INPUT statement, and the break character
  uiii be put into variable INBRCHRR. The
  end-of-file will be signalled by INEOF
  bring set to TRUE after some cai l to an
  input function has found that there is no
  more data in tho file;

WHILE NOT INEOF 00
    BEGIN
      .. code to do input -- see below. . . .
    END; .
RELEASE (INCHAN);

END;
```

5.3.2 Reading and Writing Disk Files

Most input and output will probably be done to the disk. The disk (and, typically, the DECtape) are **directory devices,** which means that logically separate files are associated with the device. When using a directory device, it is necessary to associate a **file name** with the channel that is open to the device.

```
LOOKUP(CHAN, "FILENAME", @FLAG);
ENTER(CHAN, "FILENAME", @FLAG);
```

File names are associated with channels by three functions: LOOKUP, ENTER, and RENAME. We will discuss LOOKUP and ENTER here. Both LOOKUP and ENTER take three arguments: a channel number, such as returned by GETCHAN, which has already been opened; a text string which is the name of the file, using the file name conventions of the operating system; and a reference flag that will be set to FALSE if the operation is successful, or TRUE otherwise. (The TRUE value is a bit pattern indicating the exact cause of failure, but we will not be concerned with that here.) There are three permutations of LOOKUP and ENTER that are useful:

1) LOOKUP alone: this is done when you want to read an already existing file.

2) ENTER alone: this is done when

you want to write a file. If a file already exists with the selected name, then **a** new one is created, and upon closing of the file, the old version is deleted altogether. This is the standard way to write a file.

3) A LOOKUP followed by an ENTER using the same name: this is the standard way to read and write an already existing file.

The following program will read an already existing text file, (e.g., with the INPUT, **REALIN,** and **INTIN** functions, which scan ASCII text.) Note **that** the **LOOKUP** function is used to see if the file is there, obtaining the name of the file from the user. See below for details about the functions **that** are used for the actual reading of **the data** in the file.

```
BEGIN
INTEGER INCHAN,INBRCHAR,INEOF,FLAG;
STRING F ILENAME;

OPEN (INCHAN← GETCHRN, "DSK", 8, 2, 0, 288,
      INBRCHRR, INEOF);

UHILE TRUE DO
    BEGIN
      PRINT("Input file name *");
      LOOKUP(INCHAN, FILENAME ← INCHWL, FLAG);
      IF FLAG THEN OONE ELSE
        PRINT("Cannot find file ", FILENAHE,
" try again.
");
    END;

UHILE NOT INEOF 00
   BEGIN "INPUT"
      .... see below for reading characters...
   END "INPUT";

RELEASE(INCHAN);
END;
```

The following program opens a file for writing **characters.**

```
BEGIN
INTEGER OUTCHAN, FLAG;
STR ING FILENAME;

OPEN (OUTCHAN← GETCHRN, "DSK", 0, 0, 2, 0,
      0, 0);

UHILE TRUE DO
    BEG IN
      PRINT("Output file name  *");
      ENTER(OUTCHAN, FILENAHE ← INCHUL, FLAG);
      IF NOT FLAG THEN DONE ELSE
        PRINT("Cannot write f i i r ", FILENAME,
" try again.
");
    END;

 ... now write the text to OUTCHAN . . .

RELERSE (OUTCHAN);
END;
```

5.3.2.1 Reading and Writing Full Words

Reading 36-bit **PDP10** words, using **WORDIN** and ARRYIN, and writing words using **WORDOUT** and ARRYOUT, is accomplished by opening the file using a binary mode such **as** '14. We recommend the use of binary mode, with 2 or more input and/or output buffers selected in the call to the OPEN function. There are other modes available, such **as** mode '17 for dump mode transfers; see the timesharing manual for the operating system.

5.3.2.2 Other Input /Output Facilities

Files can be renamed using the RENAME function. Some random input and output is offered by the USETI and **USETO** functions, but random input. and output produces strange results in TOPS-10 Sail. Best results are obtained by using USETI and **USETO** and reading or writing **128-word** arrays to the disk with ARRYIN and ARRYOUT.

Magnetic tape operations are performed with the MTAPE function.

See the Sail manual (Sec. 7) for more details about these functions. In particular, we stress that we have not covered all the capabilities of the functions that we have discussed.

5.3.3 TENEX Sail Channel, and File Handling

TENEX Sail has included all of the TOPS-l 0 Sail functions described in Section 7.2 of the Sail manual for reasons of **compatibility** and has implemented them suitably to work on TENEX. **Descriptions** of how these functions actually work in TENEX **are** given in Section 12.2 of the manual. However, they are less efficient than the new set of specifically TENEX routines which

have been added to TENEX Sail so you probably should skip these sections of the manual. The new TENEX routines are also greatly simplified for the user so that a number of the steps to establishing the I/O are done transparently.

Basically, you only need to know three commands: 1) OPENFILE which establishes a file on a channel, 2) SETINPUT which establishes certain parameters for the subsequent inputs from the file, and 3) CFILE which closes the file and releases the channel when you are finished.

```
chan# ← OPENFILE("filename", "modes")
```

The OPENFILE function takes 2 arguments: a string containing the device and/or filename and a string constant containing a list of the desired modes. OPENFILE returns an integer which is the channel number to be used in all subsequent inputs or outputs. If you give NULL as the filename then OPENFILE goes to the user's terminal to get the name. (Be sure if you do this that you first PRINT a prompt to the terminal.) The modes are listed in the Sail manual (Sec. 12.3) but not all of those listed are commonly used. The following are the ones that you will usually give:

- R or W or A for Read, Write, or Append depending on what you intend to do with the file.

- * if you are allowing multi-file specifications, e.g., data.*;* .

- C if the user is giving the filename from the terminal, C mode will prompt for [confirm].

- E if the user is giving the filename and an error occurs (typically when the wrong filename is typed), the E mode returns control to your program. If, E is not specified the user is automatically asked to try again.

Modes O and N for Old or New File are also allowed but probably shouldn't be used. They are misleading. The defaults, e.g. without either O or N specified, are the usual conditions (read an old version and write a new version). The O and N options are peculiar. For example, "NW" means that you must specify a completely new filename for the file to be written, e.g., a name that has not been used before. N does not mean a new version as one might have expected. In general, the I/O routines use the relevant JSYS's directly and thus include all of the design errors and bugs in the JSYS's themselves.

```
INTEGER infile,outfile,defaultsFile;
PRINT("Input file: ");
inFile ← OPENFILE(NULL,"rc");
PRINT("Output file: ");
outFile ← OPENFILE(NULL,"wc");
defaultsfile ←
    OPENFILE ("user-defaults.tmp","w");
```

We now have files "open" on 3 channels--one for reading and two for writing. We have the channel numbers stored in inFile, outFile, and defaultsFile so that we can refer to the appropriate channel for each input or output. Next we need to do a SETINPUT on the channel open for input (reading).

```
SETINPUT(chan#, count, @brchar, @eof)
```

There are four arguments:

1) The channel number.

2) An integer number which is the maximum number of characters to be read in any input operation (the default if no SETINPUT is done is 200).

3) A reference integer variable where the input function will put the break character.

4) A reference integer variable where the input function will. put true or false for whether or not the end-of-file was reached (or the error number if an error was encountered while reading).

So here we need:

```
INTEGER infileBrChr,infileEof;
SETINPUT(infile, 200, infilebrchr, infileEof);
```

Now we do the relevant input/output operations and when finished:

```
CFILE(infile);
CFILE(outfile);
CFILE(defaultsFile);
```

A simple example of the use of these routines for opening a file and outputting to it is:

```
INTEGER outfile;
PRINT("Type filename for output: ");
outfile←OPENFILE(NULL,"wc");
CPRINT(outfile, "message...");
CFILE(outfile);
```

where CPRINT is like PRINT except for the additional first argument which is the channel number.

The OPENFILE, SETINPUT, and CFILE commands will handle most situations. If you have unusual requirements or like to get really **fancy** then there are **many** variations of file handling available. A few of the more commonly used will be covered in the next section; but do not read this section until you have tried the regular routines and need to do more (if ever). On first reading, you should now skip to Section 5.4.

5.3.4 Advanced TENEX Sail Channel and File Handling

If you want to use multiple file designators with *'s, you should give "*" as one of the options to OPENFILE. Then you will need to use INDEXFILE to- sequence **through** the multiple files. The syntax is

```
found!another!file ← INDEXFILE(chan#)
```

where found!another!file is a boolean variable. INDEXFILE accomplishes two things. First, if there is another file in the sequence, it is properly initialized on the channel; and second, INDEXFILE returns TRUE to indicate that it has gotten another file. Note that the original OPENFILE gets the first file in the sequence on the channel so that you don't use the INDEXFILE until you **have** finished processing the first file and are ready for the second. This is done conveniently with a DO...UNTIL where the test is not made until after the first time through the loop; e.g.,

```
multiFilet ← OPENFILE("data.*","r*");
DO
    BEGIN
    ...<input and process current file>...
    END
    UNTIL NOT INDEXFILE(multiFiles);
```

Another **available option to the OPENFILE** routine which you should consider using is the "E" option for error handling. If you specify this option and the user gives an incorrect filename then OPENFILE will return -1 rather than **a** channel number and the TENEX error number will be returned in !SKIP!. Remember to declare EXTERNAL INTEGER !SKIP! if you are going to be looking at it. Handling the errors yourself is often **a good** idea. TENEX is unmerciful. If the user gives **a** bad filename, it will ask again and keep on asking forever even when it is obvious after a certain number of tries that there **is a** genuine problem **that needs to be resolved.**

Another use for the "E" mode is to offer the user the option of typing a bare <CR> to get a default file. If the "E" mode has been specified and the user types a carriage-return for the filename then we know that the error number returned in !SKIP! will be the number (listed in the JSYS manual) for "Null filename not allowed." so we can intercept this error and simply do another OPENFILE with the default filename, e.g.,

```
EXTERNAL INTEGER !SKIP!;
outfile←-1;
WHILE outfile= -1 DO
    BEGIN
    PRINT("Filename (<CR> for TTY:) *");
    outfile←OPENFILE(NULL,"we");
    IF !skip! = '600115 THEN
        outfile←OPENFILE("TTY:","w");
    END;
```

The GTJFNL and GTJFN routines are useful if you need more options than are provided in the OPENFILE routine, but neither of these actually opens the file so you will need an OPENF or OPENFILE after the GTJFNL or GTJFN unless your purpose in using the GTJFN is specifically that you do not want to open the file. The GTJFNL routine is actually the long form of the GTJFN JSYS; and the GTJFN routine is the short form of the GTJFN JSYS. See the TENEX JSYS manual for details.

Another use of GTJFNL is to combine filename specification from a string with filename specification from the user. This is **a** simple way to preprocess the filename from the user, i.e., to check if it is really a "?" rather than **a** filename. First, you need to declare !SKIP! and ask the user for a filename:

```
EXTERNAL INTEGER !SKIP!;
WHILE TRUE DO
    BEGIN "getfilename"
    PRINT("Type Input filename or ? :");
```

Next do a regular INTTY to get the reply into **a** string:

```
s • INTTY;
```

Then you process the string in any way that **you** choose, e.g., check if it is a "?" or some other special keyword:

```
IF s = "?" THEN BEGIN
                    givehelp;
                    CONTINUE "getfilename";
                    END;
```

If you decide it is a proper filename and want to use it then you give that string (with the break character from INTTY which will be in **!SKIP!** appended back on to the end of the string) to the GTJFNL.

```
chan#← GTJFNL (s&!SKIP!, '160080000088,
          '808106800181, NULL, NULL, NULL,
          NULL, NULL, NULL);
```

If the string ended in altmode meaning that the user wanted filename recognition then that will be done; and if the string is not enough for recognition and more **typein** is needed then the GTJFNL will ring the bell and go back to the user's terminal without the user knowing that any processing has gone on in the meantime, i.e., to the user it looks exactly like the ordinary OPENFILE. Thus the GTJFNL goes first to the **string 'that** you give it but **can** then go to the terminal if more is needed.

After the GTJFNL don't forget that you still need to OPENF the file. For reading a disk file,

```
OPENF (chan#, '448808280888) ;
```

is **a** reasonable default, and for writing:

```
OPENF (chan#, '4488001880081;
```

The arguments to GTJFNL are:

```
chan# • GTJFNL("filename", flags, jfnjfn,
               "dev", "dir", "name", "ext",
               "protection", "acct");
```

where the-flag specification is made by looking up the FLAGS for the GTJFN JSYS in the JSYS manual and figuring out which bits **you** want turned on and which off. The 36-bit resulting word can be given here in its octal representation. '160000000000 means bits 2 (old file only), 3 (give messages) and 4 (require confirm) are turned on. Remember that the bits start with Bit 0 on the left. The jfnjfn will probably always be '000100000101. This argument is for the input and output devices to be used if the string needs to be supplemented. Here the controlling terminal is used for both. Devices on the system have an octal number associated with them. The controlling terminal as input device is '100 and **as** output is '101. For most purposes you can refer to the terminal by its "name" which is TTY: but here the number is required. The input and output devices **are** given in half word format which means that '100 is in the left and '101 in the right half of the word with the appropriate O's filled out for the rest.

The next six arguments to GTJFNL are for defaults if you want to give them for: device, directory, file name, file extension, file protection, and file account. If no default is given for a field then the standard default (if any) is used, e.g., DSK: for device and Connected Directory for directory. This is another reason why you may choose GTJFNL over **OPENFILE** for getting a filename. In this way, you can set up defaults for the filename or extension. You can also use GTJFNL to simulate a directory search path. For example, the EXEC when accepting the name of a program to be run follows a search path to locate the file. First it looks on **<SUBSYS>** for a file of that name with **a .SAV** extension. Next it looks on the connected directory and finally on the **login** directory. If you have an analogous situation, you can use a hierarchical series of **GTJFNL's** with the appropriate defaults specified:

```
EXTERNAL INTEGER !SKIP!;
INTEGER logdir,condir,ttyno;
STRING logdirstr,condirstr;

GJINF(logdir,condir,ttyno);
COMMENT puts the directory numbers for login
    and connected directory and the tty# in
    its reference integer arguments1
logdirstr←DIRST(logdir);
condirstr←DIRST(condir);
COMMENT returns a string for the name
    corresponding to directory#;
WHILE true DO
  BEGIN "getname"
    PRINT("Type the name of the program: "l;
    IF EQU (upper (NAME ←INTTY),"EXEC") THEN
      BEGIN
        name←"<SYSTEM>EXEC.SAV";
        DONE "getname";
      END;
    IF name ="?" THEN
```

```
      BEGIN
        givehelp;
        CONTINUE "getname";
      END:
    name&tame8 ! SKIP!;
    COMMENT put the break char back on;
    DEFINE flag = <'100000000000>,
    jfnjfn = <'100000101>;
    IF (tempChan←GTJFNL(name,flag,jfnjfn,NULL,
        "SUBSYS",NULL,"SAV",NULL,NULL)) = -1
     THEN
      IF (tempChan←GTJFNL (name, flag,
         jfnjfn,NULL,condirstr,NULL,
         "SAV",NULL,NULL))= -1 THEN
        IF (tempChan←GTJFNL (name, flag,
          jfnjfn,NULL,    logdirstr,NULL,
          "SAV",NULL,NULL))=-1 THEN
         BEGIN
          PRINT("  ?",crlf);
          CONTINUE "getname";
         END;
   COMMENT try each default and if not found
   then try next until none are found then
   print ? and try again;
    name ← JFNS (tempChan,0);
    COMMENT gets name of file on than--8
            means in normal format;
    CFILE(tempChan);
    COMMENT channel not opened but does
        need to 'be released;
    DONE "ge tname";
    END;
```

In this case, we did not want to open a channel at all since we will not be either reading or writing the **.SAV** file. At the end of the above code, the complete filename is stored in STRING name. We might wish to run the program with the RUNPRG routine. GTJFN and GTJFNL are often used for the purpose of establishing filenames even though they are not to be opened at the moment. However, the Sail channel does need to be released afterwards.

Some of the other JSYS's which have been implemented in the **runtime package** were used in this program: GJINF, **DIRST, and** JFNS. JFNS in particular is very useful. It returns a string which is the name of the file open on the channel. You might need this name to record or to print on the terminal or because you will be outputting to a new version of the input file which you can't do unless you know its name.

These and a number of other routines are covered in Section 12 of the Sail manual. You should probably glance through and see what is there. Many of these commands correspond directly to utility JSYS's available in TENEX and will be difficult to use if you are not familiar with the JSYS's and the JSYS manual.

5.4 input from a File

In this section, we will assume that you have a file opened for reading on some channel and are ready to input. Also that you have appropriately established the end-of-file and break character variables to be used by the input routines and the break table if needed.

Another function which can be used in conjunction with the various input functions is SETPL:

```
SETPL (chan#, @line#, @page#, @sos#)
```

This allows you to set up the three reference integer variables **line#, page#,** and **sos#** to be associated with the channel so that any input function on the channel will update their values. The **line#** variable is incremented each time a '12 (lf) is input' and the **page#** variable is incremented (and **line#** reset to **0)** each time a '14 (formfeed) is input. The last SOS line number input (if any) will be in the **sos#** variable. The SETPL should be given before the inputting begins.

The major input function for text is INPUT.

```
"result"← INPUT (chan#, table #);
```

where you give **as** arguments the channel number and the break table number; and the resulting input string is returned. This is very similar to SCAN.

To input one line at **a** time from a file (where **infile** is the channel number and **infileEof** is the end-of-file variable):

```
SETBREAK(readLine←GETBREAK,lf,NULL,"ina");
  DO
    BEGIN
    STRING line;
    line←INPUT(infile,readLine);
    ....<process thr line>...
    END
  UNTIL infileEof;
```

If the INPUT function sets the eof variable to TRUE then' either the end-of-file was encountered or there was a read error of some sort.

If the INPUT terminated because a break character was read then the break character will be in the brchar variable. If brchar-0 then you have to look at the eof variable also to determine what happened: If eof=TRUE then that was what terminated the INPUT but if eof=FALSE and brchar=0 then the INPUT was terminated by reaching the maximum count per input that was specified for the channel.

If you are inputting numbers from the channel then

```
realVar ← REALIN (chan#)
integerVar ← INTIN (chan#)
```

which are like REALSCAN and INTSCAN can be used. The brchar established for the channel will be used rather than needing to give it as an argument as in the REALSCAN and INTSCAN.

INPUT is designed for files of text. Several other input functions are available for other sorts of files.

```
Number ← WORDIN (chan#)
```

will read in a 36-bit word from a binary format file. For details see the manual.

```
ARRYIN(chan#, @loc, count)
```

is used for filling arrays with data from binary format files. Count is the number of 36-bit words to be read in from the file. They are placed in consecutive locations starting with the location specified by loc, e.g.,

```
INTEGER ARRAY numbs [1:max];
ARRYIN(dataFile,numbs[1],max);
```

ARRYIN can only be used for INTEGER and REAL arrays (not STRING arrays).

5.4.1 Additional TENEX Sail input Routines

Two extra input routines which are quite fast have been added to TENEX Sail to utilize the available input JSYS's.

```
char ← CHARIN (chan#)
```

inputs a single character which can be assigned to an integer variable. If the file is at the end then CHARIN returns 0.

```
"result" ←
    SINI(chan#, maxlength, break-character)
```

does a very fast input of a string which is terminated by either reading maxlength characters or encountering the break-character. Note that the break-character here is not a reference integer where the break character is to be returned; rather it actually is the break character to be used like the "break-characters" established in a break table except that only one character can be specified. If the SINI terminated for reaching maxlength then !SKIP! = -1 else !SKIP! will contain the break character.

TENEX Sail also offers random I/O which is not available in TOPS-10 Sail. A file bytepointer is maintained for each file and is initialized to point at the beginning of the file which is byte 0. It subsequently moves through the file always pointing to the character where the next read or write will begin. In fact the same file may be read and written at the same time (assuming it has been opened in the appropriate way). If the pointer could only move in this way then only sequential I/O would be available. However, you can reset the pointer to any random position in the file and begin the read/write at that point which is called random I/O.

```
charptr ← RCHPTR (chan#)
```

returns the current position of the character pointer. This is given as an integer representing the number of characters (bytes) from the start of the file which is byte 0. You can reset the pointer by

```
SCHPTR (chan#, newptr)
```

If newptr is given as -1 then the pointer will be set to the end-of-file.

There are many uses for random I/O. For example, you can store the help text for a program in a separate file and keep track of the bytepointer to the start of each individual message. Then when you want to print out one of the messages, you can set the file pointer to the start of the appropriate message and print it out.

RWDPTR AND SWDPTR are also available for random I/O with words (36-bit bytes) as the primary unit rather than characters (7-bit bytes).

5.5 Output to a File

The CPRINT function is used for outputting to text files.

```
CPRINT (chan#, arg1, arg2, . . . . , argN)
```

CPRINT is just like PRINT except that the channel must be given as the first argument.

```
FOR i←1 STEP 1 UNTIL maxWorkers DO
    CPRINT(outfile, name[i], " ",
           salary[i],crlf);
```

Each subsequent argument is converted to a string if necessary and printed out to the channel.

```
WORDOUT (chan#, number)
```

writes a single 36-bit word to the channel.

```
ARRYOUT(chan#, @loc, count)
```

writes out an array by outputting count number of consecutive words starting at location loc.

```
REAL ARRAY results [1:max];
      .
      .
ARRYOUT(resultFile,results[1],max);
```

TENEX Sail also has the routine:

```
CHAROUT(chan#, char)
```

which outputs a single character to the channel.

The OUT function is generally obsolete now that CPRINT is available.

SECTION 6

Records

Records are the newest data structure in Sail. They take us beyond the basic part of the language, but we describe them here in the hope that they will be very useful to users of the language. Sail records are similar to those in ALGOL W (see Appendix A for the differences). Some other languages that contain record-like structures are SIMULA and PASCAL.

Records. can be extremely useful in 'setting up complicated data structures. They allow the Sail programmer: 1) a means of program controlled storage allocation, and 2) a simple method of referring to bundles of information. (Location(x) and memory [x], which are not discussed here and should be thought of as liberation from Sail, allow one to deal with addresses of things.)

6.1 Declaring and Creating Records

A record is rather like an array that can have objects of different syntactic types. Usually the record represents different kinds of information about one object. For example, we can have a class of records called person that contains records with information about people for an accounting program. Thus, we might want to keep: the person's name, address, account number, monetary balance. We could declare a record class thus:

```
RECORD!CLASS psrson (STRING name, address;
                     INTEGER account;
                     REAL ba l● ncr)
```

This occurs at declaration level, and the identifier person is available within the current block -- just like any other identifier.

RECORD!CLASS declarations do not actually reserve any storage space. instead they define a pattern or template for the class, showing what fields the-pattern has. in the above, name, addrsss, account and balance are all fields of the **RECORD!CLASS** person.

To create a record (e.g., when you get the data on an actual person) you need to call the NEW!RECORD procedure, which takes as its argument the RECORD!CLASS. Thus,

```
rp ← NEW!RECORD (person);
```

creates a person, with ail fields initially 0 (or NULL for strings, **etc).** Records are created dynamically by the program and are garbage collected when there is no longer a way to access them.

When a record is created, NEW!RECORD returns a pointer to the new record. This pointer is typically stored ii **a** RECORD!POINTER. **RECORD!POINTERs** are **variables** which must be declared. The RECORD!POINTER rp was used above. There is a very important distinction to be made between a RECORD!POINTER and a RECORD. A RECORD is a block of variables called fields, and a RECORD!POINTER is an entity that points to some RECORD (hence can be thought of as the "name" or "address" of a RECORD)., A RECORD has fields, but a RECORD!POINTER does not, although its associated RECORD may have fields. The following is a complete program that **declares a** RECORD!CLASS declares **a** **RECORD!POINTER,** and creates 'a record in **the** RECORD!CLASS with the pointer to the new record stored in the RECORD!POINTER.

```
BEGIN
RECORD!CLASS person (STRING name,address;
                     INTEGER account;
                     REAL balance);
RECORD!POINTER (person) rp;

COMMENT program starts hero.;
rp ← NEW!RECORD (person) ;
END;
```

RECORD!POINTERs are usually associated with particular record **class(es).** Notice that in the above program the declaration of RECORD!POINTER mentions the class psrson:

```
RECORD!POINTER (person) rp;
```

This means that the compiler will do type checking and make sure that only pointers to records of class person will be stored into rp. A **RECORD!POINTER** can **be** of several classes, as in:

```
RECORD!POINTER (person, university) rp;
```

assuming that we had a RECORD!CLASS university.

RECORD!POINTERs can be of any class if we say:

```
RECORD!POINTER (ANYICLASS) rp;
```

but declaring the **class(es)** of record pointers gives compilation time checking of record class agreement. This becomes an advantage when you have several classes, since the compiler will complain about many of the simple mistakes you can make by mis-assigning record pointers.

6.2 Accessing Fields of Records

The fields of records can be read/written just like the elements of arrays. Developing the above program a bit more, suppose we have created a new record of class **person,** and stored the pointer to that record in rp. Then, we can give the "person" a name, address, etc., with the following statements.

```
person:name[rp] ← "John Doe";
person: address [rp]← "181 East Lansing Street";
person: account [rp]← 14;
person: balance [rp]← 3888.871
```

and we could write these fields out with the statement:

```
PRINT ("Name is ", person:name[rp], crlf,
      "Address is ", person:address[rp], crlf,
      "Account is ", person:account[rp], crlf,
      "Balance is ", person:balance[rp], crlf);
```

The syntax for fields has the following features:

1) The fields are available within the lexical scope where the RECORD!CLASS was declared, and follow ALGOL block structure.

2) The fields in different classes may have the same name, e.g., parent:name and child: name.

3) The syntax is rather like that for arrays -- using brackets to surround the record pointer in the same way brackets are used for the array index.

4) The fields can be read or written into, also like array locations.

5) It is necessary to write class: field[pointer] -- i.e., you have to include the name of the class (here **person**) with a ":" before the name of the field.

6.3 Linking Records Together

Notice, in the above example, that as we create the persons, we have to store the pointers to the records somewhere or else they will become "missing persons". One way to do this would be to use an array of record pointers, allocating as many pointers as we expect to have people. If the number of people is not known in advance then the more customary approach is to link the records together, which is done by using additional fields in the records.

Suppose we upgrade the above example to the following:

```
RECORD!CLASS person WRING name, address;
             INTEGER account;
             REAL balance;
             RECORD!POINTER(ANY!CLASS) next);
```

Notice now that there is a RECORD!POINTER field in the template. This may be used to keep a pointer to the next person. The header to the entire- list of persons will be kept in a single **RECORD!POINTER.**

Thus, the following program would create persons dynamically and put them into a "linked list" with the newest at the head of the list. This technique allows you to write programs that are not restricted to some fixed maximum number of persons, but instead allocate the memory space necessary for a new person when you need it.

```
BEGIN
RECORO!CLRSS person (STRING name, address;
    INTEGER account; RERL balance;
    RECORD!POINTER(ANY!CLASS) next);

RECORD!POINTER (ANY!CLASS) header;

WHILE TRUE 00
  BEGIN
    STRING s;
    RECORD!POINTER (ANY!CLASS) tamp;

    PRINT("Name of next person, CR if donor");
    IF NOT LENGTH(s←INCHWL) THEN DONE;

    COMMENT put neu person at head of list;
    temp . NEW!RECORD(person);
    COMMENT make a neu record;
    person: next[temp] ← header;
    COMMENT the old head becomes the second;
```

```
        header ← tamp;
        COMMENT the n e w record becomes the head;
        COMMENT now fill informatlon fields;
        person: name [temp] ← s;
        COMMENT nou we can fill address, account,
       . balance if we want...;
    END;

END;
```

A very powerful feature of record structures is the ability to have different sets of pointers. For example, there might be both forward and backward links (in the above, we used a forward link). Structures such as binary trees, sparse matrices, deques, priority queues, arid so on are natural applications of records, but it will take a little study of the structures in order to understand how to build them, and what they are good for.

Be warned about the difference. between records, record pointers, record classes, and the fields of records: they are all distinct things, and you can get in trouble if you forget it. Perhaps a simple example will show you what is meant:

```
BEGIN
RECORD!CLASS pair (INTEGER i,j);
RECORD!POINTER (pair) a, b, c, d;

a ← NEW!RECORD (pair);
pair: i [a] ← l;
pair:) [a] ← 2 ;
d ←
b ← NEW!RECORD (pair);
pair: i [b] ← 1 ;
pair:j[b] ← 2;
c ← NEW!RECORD (pair);
pair: i [c] ← 1;
pair: j [c] ← 3;
IF a = b THEN  PRINT("A = B ");
pair: j [d] ← 3 ;
IF a = c THEN PRINT(" A = C ");
IF c = d THEN PRINT(" C = 0 ");
IF a = d THEN PRINT(" R = 0 ");
PRINT( "(A I : " , pair: i [a], ", J:",
       pair: j [a], ")" );
PRINT( "(B I:", pair; i [b], ", J:",
       pair: j [b], ")");
PRINT( " (C I t", pair: i [c], ", J:",
       pair: j [c], ")");
PRINT( " (D I:", pair: i [d], ", J:",
       pair: j [d], ")" );
END;
```

will print:

```
A = 0 (A I:1, J:3) (B I:1, J:2)
(C I:1, J:3) (D I:1, J:3)
```

Note that two RECORD!POINTERs are only equal if they point to the same record (regardless of whether the fields of the records that they point to are equal). At the end of executing the previous example, there are 3 distinct records, one pointed to by RECORD!POINTER **b,** one pointed to by RECORD!POINTER c, and one pointed to by RECORD!POINTERs a and **d.** When the line that reads: `pair:j[d] ← 3;` is executed, the j-field of the record pointed at by RECORD!POINTER d is changed to 3, not the j-field of **d** (RECORD!POINTERs have no fields). Since that is the same record as the one pointed to by RECORD!POINTER a, when we print `pa irr j [a]`, we get the value 3, not 2.

Records can also help your programs to be more readable, by using a record as a means of returning a collection of values from a procedure (no Sail procedure can return more than one value). If you wish to return a RECORD!POINTER, then the procedure declaration must indicate this as an additional type-qualifier on the procedure declaration, for example:

```
RECORD!POINTER (person) PROCEOURE maxBalance;
BEGIN
RECOROIPOINTER (person) tempHeader,
                        currentMaxPerson;
RERL currentllaxl
tempHeader ← header;
currentMax ← personrbalance [tempHeader];
currentMaxPerson ← tempHeader;
WHILE tempHeader ← personrnext [tempHeader] 0 0
IF personr balance [tempHeader] > currentMax THEN
  BEGIN
    currentMax ← person:balance [tempheader];
    currentMaxPerson ← tempHeader;
  END;
RETURN(currentMaxPerson);
END;
```

This procedure goes through the linked list of records and finds the person with the highest balance. It then returns a record pointer to the record of that person. Thus, through the single RETURN statement allowed, you get both the name of the person and the balance.

RECORD!POINTERs can also be used as arguments to procedures; they are by default VALUE parameters when used. Consider the following quite complicated example:

```
RECORD!CLASS pnt (REAL x,y,z);
RECORD!POINTER (pnt) PROCEDURE midpoint
                (RECORD!POINTER (pnt) a,b) ;
```

```
BEGIN
RECORD!POINTER(pnt) retval;
retval ← NEW!RECORD(pnt);
pnt:x [retval] ← (pnt:x [a] + pnt:x [b]) / 2;
pnt:y [retval] ← (pnt:y [a] + pnt:y [b]) / 2;
pnt:z [retval] ← (pnt:z [a] + pnt:z [b]) / 2;
RETURN(retval);
END;

...
p ← midpoint(q, r);
...
```

While this procedure may appear a bit clumsy, it makes it easy to talk about such things as **pnts** later, using simply a record pointer to represent each pnt. Another common method for "returning" more than one thing from a procedure is to use REFERENCE parameters, as in the following example:

```
PROCEDURE midpoint (REFERENCE REAL rx,ry,rz;
                    REAL ax,ay,az,bx,by,bz);
BEGIN
rx ← (ax + bx) / 2;
ry ← (ay + by) / 2;
rz ← (az + bz) / 2;
END;
...
MIDPOINT(px, py, pz, qx, qy, qz, rx, ry, rz,);
...
```

Here the code for the procedure looks quite simple, but there are so many arguments to it that you can easily get lost in the main code. Much of the confusion comes about because procedures simply cannot return more than one value, and the record structure allows you to return the name of a bundle of information.

SECTION 7

Conditional Compilation

Conditional compilation is available so that the same source file can be used to compile slightly different versions of the program for different purposes. Conditional compilation is handled by the scanner in a way similar to the handling of macros. The text of the source file is manipulated before it is compiled. The format is

```
IFCR boo loan THENC code ELSEC code ENOC
```

This construction is not a statement or an expression. It is not followed by a semi-colon but just appears at any point in your program. The ELSEC is optional. The ENDC · must be included to mark the end but no begin is used. The code which follows the THENC (and ELSEC if used) can be any valid Sail syntax or fragment of syntax. As with macros, the scanner is simply manipulating text and does not check that the text is valid syntax.

The boolean must be one which has a value at compile time. This means it cannot be any value computed by your program. Usually, the boolean will be DEFINE'd by a macro. For example:

```
DEFINE smallVersion = <TRUE>;
   . . .
IFCR smallVersion THENC max ←10*total;
  ELSEC max ←100*total; ENOC
   . . .
```

where every.difference in the program between the small and large versions is handled with a similar IFCR...THENC...ENDC construction. For this construction, the scanner checks the value of the boolean; and if it is TRUE, the text following THENC is inserted in the source being sent to the inner compiler--otherwise the text is simply thrown away and the code following the ELSEC (if any) is used. Here the code used for the above will be max ←10*total;, and if you edit the program and instead

```
DEFINE smallVersion = <FALSE>;
```

the result will be max ←100*total;.

The code following the THENC and ELSEC will be taken exactly as is so that statements which need final semi-colons should have them. The above format of statement ; ELSEC is correct.

If. this feature were not available then the following would have to be used:

```
BOOLEAN smallVersion;
smallVersion • TRUE;
   • ♂⊠
IF smallVersion THEN max • 10*total
ELSE ○♋⊠ • 100*total;
   ...
```

so that a conditional would actually appear, in your program.

Some typical uses of conditional compilation are:

1) Insertion of debugging or testing code for experimental versions of a program and then removal for the final version. Note that the code will still be in your source file and can be turned back on (recompilation is of course required) at any time that you again need to debug. When you do not turn on debugging, the code completely disappears from your program but not from your source file.

2) Maintainence of a single source file for a program which is to be exported to several sites with minor differences.

```
DEFINE sumex = <TRUE>,
       isi = <FALSE>;
   . ♂ ♂
IFCR sumex THENC docdir ←"DOC"; ENOC
IFCR isi THENC docdir ←"DOCUMENTATION"; ENOC
   ...
```

where only one site is set to TRUE for each compilation.

3) "Commenting out" large portions of the program. Sometimes you need to temporarily remove a large section of the program. You can insert the word COMMENT preceding every statement to be removed but this is a lot of extra work. A better way is to use:

```
IFCR FALSE THENC
  ...
<all the code to be "removed">
  ...
ENOC
```

SECTION 8

Systems Building in Sail

Many new Sail users will find their first Sail project involved with adding to an already-existing system of large size that has been worked on by many people over a period of years. These systems include the speech recognition programs at Carnegie-Mellon, the hand-eye software at Stanford AI, large CAI systems at Stanford IMSSS, and various medical programs at SUMEX and NIH. This section does not attempt to deal with these individual systems in any detail, but instead tries to describe some of the features of Sail that are frequently used in systems building, and are common to all these systems. The exact documentation of these features is given elsewhere; this is intended to be a guide to those features.

The Sail language itself is procedural, and this means that programs can be broken down into components that represent conceptual blocks comprising the system. The block structuring of ALGOL also allows for local variables, which should be used wherever possible. The first rule of systems building is: break the system down into modules corresponding to conceptual units. This is partly a question of the design of the system--indeed, some systems by their very design philosophy will defy modularity to a certain extent. As a theory about the represent at ion of knowledge in computer programs, this may be necessary; but programs should, most people would agree, be as modular "as possible".

Once modularized, most of the parts of the system can be separate files, and we shall show below how this is possible. Of course, the modules will have to communicate together, and may have to share common data (global arrays, flags, etc.). Also, since the modules will be sharing the same core image (or job), there are certain Sail and timesharing system resources that will have to be commonly shared. The rules to follow-here are:

1) Make the various modules of a system as independent and separate as design philosophy allows.

2) Code them in a similar "style" for readability among programmers.

3) Make the points of interface and communication between the programs as clear and explicit as possible.

4) Clear up questions about which modules govern system resources (Sail and the timesharing system), such as files, terminals, etc. so that they are not competing with each other for these resources.

8.1 The Load Module

The most effective separation of modules is achieved through separate compilations. This is done by having two or more separate source files, which are compiled separately and then loaded together. Consider the following design for an AI system QWERT. QWERT will contain three modules: a scanner module XSCAN, a parser module PARSE, and a main program QWERT. We give below the three files for QWERT.

First, the QWERT program, contained in file QWERT.SAI:

```
BEGIN"QWERT"

EXTERNAL STRING PROCEDURE XSCAN(STRING S);
REQUIRE "XSCAN" LOAD!MODULE;

EXTERNAL STRING PROCEDURE PARSE(STRING S);
REQUIRE "PARSE" LOAD!MODULE;

WHILE TRUE DO
    BEGIN
        PRINT("*",PARSE(XSCAN(INCHWL)));
    END;

END"QWERT";
```

Notice two features about QWERT.SAI:

1) There are two EXTERNAL declarations. An EXTERNAL declaration says that some identifier (procedure or variable) is to be used in the current program, but it will be found somewhere else. The EXTERNAL causes the compiler to permit the use of the identifier, as requested, and then to issue a request for a global fixup to the LOADER program.

2) Secondly, there are two REQUIRE ... LOAD!MODULE statements in the program. A load module is a file that is loaded by the loader, presumably the output of some compiler or assembler. These REQUIRE statements cause the compiler to request that the loader load modules XSCAN.REL and PARSE.REL when we load MAIN.REL. This will hopefully satisfy the global requests: i.e., the loader will find the two procedures in the two mentioned files, and link the programs all together into one "system".

Second, the code for modules XSCAN and PARSE:

```
ENTRY XSCAN;
BEGIN

INTERNRL STRING PROCEDURE XSCAN(STRING S);
BEGIN
  ..... code for XSCAN ....
RETURN (resulting string);
END;

END;
```

and now PARSE.SAI:

```
ENTRY PARSE;
BEGIN

INTERNRL STRING PROCEDURE PARSE(STRING S);
BEGIN

  ....code for PARSE....
RETURN(resulting string);
END;

END;
```

Both of these modules begin with an ENTRY declaration. This has the effect of saying that the prdgram to be compiled is not a "main" program (there can be only one main program in a core image), and also says that PARSE is to be found as an INTERNAL within this file. The list of tokens after the ENTRY construction is mainly used for LIBRARYs rather than LOAD!MODULEs, and we do not discuss the difference here, since LIBRARYs are not much used in system building due to the difficulty in constructing them.

A few important remarks about LOAD!MODULES:

1) The use of LOAD!MODULES depends on the loaders (LOADER and LINK10) that are available on the system. In particular, there is no way to associate an external symbol with a particular LOAD!MODULE.

2) The names of identifiers are limited to six characters, and the character set permissible is slightly less than might be expected. The symbol "!" is, for example, mapped into "." in global symbol requests.

3) The "semantics" of a symbol (e.g., whether the symbol names an integer or a string procedure) is in no way checked during loading.

Initialization routines in a LOAD!MODULE can be performed automatically by including a REQUIRE ... INITIALIZATION procedure. For example, suppose that INIT is a simple parameterless, valueless procedure that does the initialization for a given module:

```
SIMPLE PROCEDURE INIT;
BEGIN
  ...initialization code...
END;

REQUIRE INIT INITIALIZATION;
```

will run INIT prior to the outer block of the main program. It is difficult to control the order in which initializations are done, so it is advisable to make initializations that do not conflict with each other.

8.2 Source Files

In addition to the ability to compile programs separately, Sail allows a single compilation to be made by inserting entire files into the scan stream during compilation. The construction:

```
REQUIRE "FILENM.SAI" SOURCE!FILE;
```

inserts the text of file FILENM.SAI into the stream of characters being scanned--having the same effect that would be obtained by copying all of FILENM.SAI into the current file.

One pedestrian use of this is to divide a file into smaller files for easier editing. While this can be convenient, it can also unnecessarily fragment a program into little pieces without purpose.

There are, however, some real purposes of the SOURCE!FILE construction in systems building. One use is to include code that is needed in several places into one file, then "REQUIRE" that file in the places that it is needed. Macros are a common example. For example, a file of global definitions might be put into a file MACROS.SAI:

```
REWIRE "<><>" DELIMITERS;
DEFINE ARRAYSIZE=<100>,
        NUMBEROFSTUDENTS=<200>,
        FILENAME=<"FIL.DAT">;
```

A common use of source files is to provide a SOURCE!FILE that links to a load module: the source file contains the EXTERNAL declarations for the procedures (and data) to be found in a module, and also requires that file as a load module. Such a file is sometimes called a "header" file. Consider the file XSCAN.HDR for the above XSCAN load module:

```
EXTERNAL STRING PROCEDURE XSCAN(STRING S);
REQUIRE "XSCAN" LOAD!MODULE;
```

The use of header files ameliorates some of the deficiencies of the loader: the header file can, for example, be carefully designed to contain the declarations of the EXTERNAL procedures and data, reducing the likelihood of an error caused by misdeclaration. Remember, if you declare:

```
INTERNAL STRING PROCEDURE XSCAN(STRING S);
BEGIN ..... END;
```

in one file and

```
EXTERNAL INTEGER PROCEDURE XSCAN(STRING S);
```

in another, the correct linkages will not be made, and the program may crash quite strangely.

8.3 Macros and Conditional Compilation

Macros, especially those contained in global macro files, can assist in system building. Parameters, file names, and the like can be "macroized".

Condition-al compilation also assists in systems building by allowing the same source files to do different things depending on the setting of switches. For example, suppose a file FILE is being used for both a debugging and a "product ion" version of. the same module. We can include a definition of the form:

```
DEFINE DEBUGGING=<FALSE>;
COMMENT false if not debugging;
```

and then use it

```
IFCR DEBUGGING THENC
   PRINT("Now at PROC PR ",I," ",J,CRLF); ENDC
```

(See Section 7 on conditional compilation for more details.) In the above example, the code will define the switch to be FALSE, and the PRINT statement will not be compiled, since it is in the FALSE consequent of an IFCR . ..THENC. In using switches, it is common that there is a default setting that one generally wants. The following condition al compilation checks to see if DEBUGGING has already been defined (or declared), and if not, defines it to be false. Thus the default is established.

```
IFCR NOT DECLARATION(DEBUGGING) THENC
    DEFINE DEBUGGING=<FALSE>; ENDC
```

Then, another file, inserted prior to this one, sets the compilation mode to get the DEBUGGING version if needed.

Macros and conditional compilation also allow a number of complex compile-time operations, such as building tables. These are beyond our discussion here, except to note that complex macros are often used (overused?) in systems building with Sail.

APPENDIX A

Sail and ALGOL W Comparison

There are many variants of ALGOL. This Appendix will cover only the main differences between Sail and ALGOL W.

The following are differences in terminology:

ALGOL W		Sail
I -	Rssignment operator	←
* *	Exponentiation operator	↑
¬=	Not equal	≠ or NEQ
<=	Less than or equal	≤ or LEQ
>=	Greater than or equal	≥ or GEQ
REM	Division remainder operator	MOD
END.	Program end	END
RESULT	Procedure parameter type	REFERENCE
str(i|j)	Substrings	str[i+1 for j]
STRING(i) s	String declarations	STRING s
arry (1)	Array subscript	arry [1]
arry (1: : 10)	Array declaration	arry tl: 10]

The following are not available in Sail:

ODD ROUND ENTIER

TRUNCATE Truncation is default conversion.

WRITE, WRITEON Use PRINT statement for both.

REROON Use INPUT, RERLIN, INTIN.

Block expressions

Procedure expressions
Use RETURN statement in procedures.

Other differences are:

1) Iteration variables and Labels must be declared in Sail, but the iteration variable is more general since it can be tested after the loop.

2) STEP UNTIL cannot be left out in the FOR-statement -in Sail.

3) Sail strings do not have length declared and are not filled out with blanks.

4) EQU not = is used for Sail strings.

5) The first case in the CASE statement in Sail is 0 rather than 1 as in ALGOL W. (Note that Sail also has CASE expressions.)

6) <, =, and > will not work for alphabetizing Sail strings. They are arithmetic operators only.

7) ALGOL W parameter passing conventions vary slightly from Sail. The ALGOL W RESULT parameter is close to the Sail REFERENCE parameter, but there is a difference, in that the Sail REFERENCE parameter passes an address, whereas the ALGOL W RESULT parameter creates a copy of the value during the execution of the procedure.

8) A FORWARD PROCEDURE declaration is needed in Sail if another procedure calls an as yet undeclared procedure. Sail is a one-pass compiler.

9) Sail uses SIMPLE PROCEDURE, PROCEDURE, and RECURSIVE PROCEDURE where ALGOL has only PROCEDURE (equivalent to Sail's RECURSIVE PROCEDURE).

10) Scalar variables in Sail are not cleared on block entry in non-RECURSIVE procedures.

11) Outer block arrays in Sail must have constant bounds.

12) The RECORD syntax is considerably different. See below.

Sail features (or improvements) not in ALGOL W:

a) Better string facilities with more flexibility.

b) More complete RECORD structures.

c) Use of DONE and CONTINUE statements for easier control of loops.

d) Assignment expressions for more compact code.

e) Complete I/O facilities.

f) Easy interface to machine instructions.

The following compares Sail and ALGOL W records in several important aspects.

Aspect	Sail	ALGOL W
Declaration of class	RECORD!CLASS	RECORD
Declaration of record pointer ,	RECORD!POINTER	REFERENCE
	Pointers can be several classes or ANY!CLASS	pointers must be to one class
Empty record	Reserved uord NULL!RECORD	Reserved uord NULL
Fields of record	Use brackets	Use parens
	llust use CLRSS: brfore the field name	Don' t use class name before field

REFERENCES

1. Reiser, John (ed.), Sail, Memo AIM-289, Stanford Artificial Intelligence Laboratory, August 1976.

2. Frost, Martin, UUO Manual (Second Edition), Stanford Artificial Intelligence Laboratory Operating Note 55.4, July 1975.

3. Harvey, Brian (M. Frost, ed.), Monitor Command Manual, Stanford Artificial Intelligence Laboratory Operating Note 54.5, January 1976.

4. Feldman, J.A., Low, J.A., Swinehart, D.C., Taylor, R.H., "Recent Developments in Sail", AFIPS FJCC 1972, p. 1193-1202.

5. DECSYTEM10 Assembly Language Handbook (3rd Edition), Digital Equipment Corporation, Maynard, Massachusetts, 1973.

6. DECSYSTEM10 Users Handbook (2nd Edition), Digital Equipment Corporation, Maynard, Massachusetts, 1972.

7. Myer, Theodore and Barnaby, John, TENEX EXECUTIVE Manual (revised by William Plummer), Bolt, Beranek and Newman, Cambridge, Massachusetts, 1973.

8. JSYS Manual (2nd Revision), Bolt, Beranek and Newman, Cambridge, Massachusetts, 1973.

INDEX

!SKIP! 30

& 12

ALGOL 48
allocation 15
Altmode 30
ANY!CLASS 41
Arguments 20
array 4, 7
arrays 15, 16, 38
ARRCLR '15
ARRYIN 33, 38
ARRYOUT 33, 39
assignment expressions 10
assignment operator 10
Assignment statements 5

BEGIN 2
binary format files 38
bits 36
block 2
block name 14
blocks 9, 13
BOOLEAN 2
boolean expression 8
break character 27, 30, 38
break tables 27
built-in procedures 6, 19

CASE expressions 11
CFILE 34
channel 34, 37
channel number 31
CHARIN 38
CHAROUT 39
Commenting' 44
compile time 15
compound statement 9
Conditional compilation 44
condition al expressions 11
conditionals 7
connected directory 36
constants 3
CONTINUE - 18
control statements 7
controlling-terminal 30, 36
CPRINT 39
crlf 30'
CVD 6

data 38
deallocation 15
debugging 44
Declarations 2
DEFINE 25
delimiters 25
directory devices 31, 32
DIRST 37
DO...UNTIL 17
DONE 18
dynamic 15 ,

ELSEC 44
emulator 1
END 2
end-of-file 37, 38 ,
ENDC 44
ENTER 32
ENTRY 46
eol 30
EQU 8, 11 ,
equality 8
error handling 35
expression 5, 6
expressions 10
EXTERNAL 30, 45

FALSE 2
fields 40
file bytepointer 38
file name 32
files 30
flag specification 36
FOR statement 15
format 4
FORWARD 21
free format 4

garbage collections 12
GETBREAK 2 7
GETCHAN 31
GJINF 37
global 14
GTJFN 35
GTJFNL 35

half word format 36

I/O 30
identifiers 3
IF..THEN statement 7
IFCR 44
INCHWL 6, 30
indefinite iteration 17

INDEXFILE 35
initialization 15
Initialization routines 46
INPUT 27, 37
input/output 30, 31
INTEGER 2
INTIN 38
INTSCAN 29
INTTY 30
iteration variable 16

JFNS 37

LENGTH 12
line terminators 28
line-editing 30
LOAD!MODULE 45
LOADER 45
local 14
login directory 36
LOOKUP 32
LOP 12
lowercase 4

macro expansion 25
macros 25
modularity 45
MTAPE 33
multi-dimensioned arrays 4
multiple file designators 35

nested 9, 14
NEW!RECORD 4.0
NUL character 13
NULL 3

octal representation 36
OPEN 31
OPENFILE 34
order of evaluation 10
outer block 2
OWN 15

PA1050 1
parallel arrays 4
parameter list 20
parameterized procedure 20
parent hesized 11
predeclared identifiers 3
PRINT 6
PRINT statement 25
procedure 19
procedure body 21
procedure call 19

random I/O 38
RCHPTR 38
read error 37
REAL 2
REALIN 38
REALSCAN 29
RECORD!CLASS 40
RECORD!POINTER 40
Records 40
RECURSIVE 15, 21
REFERENCE 24
reinitialization 15
RELEASE 32
RENAME 32
reserved words 2, 3
RETURN statement 21
runtime 15

scalar variables 15
SCAN 27
scanner 25
SCHPTR 38
scope of the variable 14
search path 36
semi-colon 8
sequential I/O 38
SETBREAK 27
SETFORMAT 13
SETINPUT 34
SETPL 37
SETPRINT 30
side-effect 23
SIMPLE 21
SINI 38
SOS line numbers 28
SOURCE!FILE 47
SQRT 6
Statements 2
statements 5
Storage allocation 15
STRING 2
string descriptor 12
STRING operators 11
string space 12
strings 27
subscripts 5
substrings 12

tables 13
Teletype I/O 30
TENEX Sail 1
THENC 44
TOPS-10 Sail 1
TRUE 2
TTY: 36
type conversion 6

typed procedures 22

untyped procedures 22
uppercase 4, 20, 28, 30
USETI 3 3
USETO 3 3

VALUE 24
variables 3, 14

WHILE...DO 1 7
WORDIN 33, 38
WORDOUT 33, 39