# M T S

## The Michigan Terminal System

## Volume 16: ALGOL W in MTS

September 1980

The University of Michigan Computing Center
Ann Arbor, Michigan

**********
*
*  This obsoletes the August 1978 edition.  *
*
**********

DISCLAIMER

The MTS Manual is intended to represent the current state of the Michigan Terminal System (MTS), but because the system is constantly being developed, extended, and refined, sections of this volume will become obsolete. The user should refer to the Computing Center Newsletter, Computing Center Memos, and future Updates to this volume for the latest information about changes to MTS.

Copyright 1980 by the Regents of the University of Michigan. Copying is permitted for nonprofit, educational use provided that (1) each reproduction is done without alteration and (2) the volume reference and date of publication are included. Permission to republish any portions of this manual should be obtained in writing from the Director of the University of Michigan Computing Center.

# PREFACE

The software developed by the Computing Center staff for the operation of the high-speed processor computer can be described as a multiprogramming supervisor that handles a number of resident, reentrant programs. Among them is a large subsystem, called MTS (Michigan Terminal System), for command interpretation, execution control, file management, and accounting maintenance. Most users interact with the computer's resources through MTS.

The MTS Manual is a series of volumes that describe in detail the facilities provided by the Michigan Terminal System. Administrative policies of the Computing Center and the physical facilities provided are described in a separate publication entitled Introduction to the Computing Center.

The MTS volumes now in print are listed below. The date indicates the most recent edition of each volume; however, since volumes are updated by means of CCMemos, users should check the Memo List, copy the files *CCMEMOS or *CCPUBLICATIONS, or watch for announcements in the Computing Center Newsletter, to ensure that their MTS volumes are fully up to date.

Volume 1: The Michigan Terminal System, December 1979
Volume 2: Public File Descriptions, December 1978
Volume 3: System Subroutine Descriptions, October 1976
Volume 4: Terminals and Tapes, August 1974
Volume 5: System Services, April 1980
Volume 6: FORTRAN in MTS, December 1978
Volume 7: PL/I in MTS, July 1977
Volume 8: LISP and SLIP in MTS, June 1976
Volume 9: SNOBOL4 in MTS, September 1975
Volume 10: BASIC in MTS, September 1974
Volume 11: Plot Description System, August 1978
Volume 12: PIL/2 in MTS, December 1974
Volume 14: 360/370 Assemblers in MTS, August 1978
Volume 15: FORMAT and TEXT360, April 1977
Volume 16: ALGOL W in MTS, September 1980

Other volumes are in preparation. The numerical order of the volumes does not necessarily reflect the chronological order of their appearance; however, in general, the higher the number, the more specialized the volume. Volume 1, for example, introduces the user to MTS and describes in general the MTS operating system, while Volume 10 deals exclusively with BASIC.

The attempt to make each volume complete in itself and reasonably independent of others in the series naturally results in a certain amount of repetition. Public file descriptions, for example, may appear in more than one volume. However, this arrangement permits the user to buy only those volumes that serve his or her immediate needs.

Richard A. Salisbury

General Editor

# PREFACE TO VOLUME 16

Algol W is a general-purpose programming language which is a development of Algol 60. The main differences concern:

- Character handling
- Complex arithmetic
- A more convenient 'while' construct
- A simpler 'for' loop
- The 'case' construct
- Data structures (using 'record' and 'reference')

Note that Algol W is not a superset of Algol 60; Algol 60 programs cannot be compiled and run in Algol W without considerable editing.

This manual describes the Algol W language, the Algol W compiler, and aspects of MTS relevant to the use of the Algol W compiler. It is intended for use as a reference manual. It is not meant for teaching programming concepts. The text does not assume a great deal of programming experience, and thus is appropriate for even the beginner.

Blanche Grosswald wrote the major portion of this volume. She is grateful to members of the University of Michigan Computing Center staff and to people in the department of Computer and Communication Sciences at the University of Michigan, who read drafts of this volume, offered suggestions and examples, and pointed out ways to clarify many topics.

Alan Hunter revised the text in the light of developments to Algol W at the University of Newcastle upon Tyne. This has added sections on the input/output system, external subroutine linkages, other new predeclared procedures, and new features provided by the compiler. The main text describing the Algol W language has been modified only very slightly. NUMAC wishes to thank the University of Michigan for the use of their MTS Volume 16 text, and the University of Alberta for use of other material.

The system changes for this version of Algol W were designed and coded by Alan Hunter, Margaret Hindmarsh (of Newcastle University Computing Laboratory), and James Bodwin (of the University of Michigan Computing Center).

Thanks are due to members of staff of several MTS installations, in particular James Eve, Peter King and John Lloyd (of Newcastle); George Helffrich (of Michigan); Kathryn Ward, Tony Marsland and Antoine Verheijen (of the University of Alberta). Without their help, advice, and encouragement, enhancements to the Algol W system would not have been possible.

Contents

# INTRODUCTION TO ALGOL W

## TERMINOLOGY

### Notation for Describing the Algol W Language

The symbols < > are used throughout this manual to enclose generic terms to be replaced by an item supplied by the user.

### Basic Symbols in Algol W

Algol W programs are written using the following basic character set:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

a b c d e f g h i j k l m n o p q r s t u v w x y z

0 1 2 3 4 5 6 7 8 9

+ - * / = : ; . , # " ' | ¬ ( ) < > % _

Certain groups of letters form 'reserved words'. Examples of these are 'begin' and 'end', introduced later in this chapter. Reserved words cannot be used as identifiers (see the section "Identifiers") in an Algol W program. A list of reserved words is in Appendix D.

Note that in reserved words and identifiers, described later, lower case letters are treated as if they were the equivalent upper case letter.

The symbol <empty>, where <empty> is no physical symbol, is also considered an element of the Algol W set of symbols.

### Format

Reserved words, identifiers, and constants (see the section "Constants, Variables, Expressions, and Values") must not include blanks and must be separated from each other by at least one blank or special character (other than underscore) when appropriate. Otherwise, blanks have no meaning and can be used freely throughout an Algol W program.

For example, blanks are not required to separate any of the special characters from an identifier, a number, or each other.

Lines composing Algol W programs can be punched or typed anywhere in the first 255 columns. It is not necessary to begin a line in a given column. Reserved words, identifiers, and constants may not cross a line boundary. Data items may be placed anywhere in the first 256 columns. In simple input, no data item may cross a line boundary.

## Presentation of Examples in this Manual

Algol W programs may be entered in any mixture of upper and lower case text, because the implementation is case independent. To aid readability within this manual, examples of Algol W program text are formatted in the manner which the compiler would list them if the compiler parameters TRIDENTIFIER=MIXEDCASE, TRRESERVED=LOWERCASE and INDENT were specified (see the section "Algol W Programmer's Guide"). This implies:

(1) Reserved words are presented in lowercase text. For example:

```
array
begin
reference
```

(2) Identifiers are also presented in lower case text, but with the first letter and any letter immediately following an underscore character capitalised. For example:

```
I
Score
Number_Of_Elements
```

(3) The indenting of source text in the examples is that which the compiler would produce if the indent option were specified and if the text were not indented in the source input. The exceptions to this rule are in some comments and statement continuations where extra indentation has been used for clarity. For example:

```
begin
   integer Score, Item;
   Read(Score, Item)
end.
```

There is, therefore, no need to input program source text in mixed case; uppercase only punched cards and mixed case file lines are equally acceptable to the compiler.

Within the text of this reference manual, reserved words are usually presented in lower case enclosed in primes (’). For example:

...’if’ and ’case’ are reserved words ...

It should be clearly understood that this notation is to avoid ambiguity in the English of the manual text; reserved words are never enclosed in primes within source programs. Identifiers are distinguished in the manual text by the capitalization of the first character of the name.

## BLOCK STRUCTURE

Almost all Algol W programs have the form:

<statement>.

where:

<statement> can be any legal Algol W statement (see the section "Statements," for other program forms and for the definition of <statement>), but usually is a block; and

the period (.) is part of the program.

A block starts with the reserved word ’begin’ and ends with the reserved word ’end’. In between, it contains two types of information:

(1) declarations which describe variables used within the block, and

(2) statements which describe the action of the block.

The format of a program block is:

```
begin
   <declarations>
   <statements>
end.
```

Declarations specify the types of given variables (see the section "Values and Types") and statements perform actions on the variables. Each declaration and statement must be separated from the following declaration or statement by a semicolon (;). The semicolon is used to separate Algol W declarations and statements, not to terminate them. Since ’begin’ and ’end’ are examples of Algol W punctuation, and are not statements or declarations, there is no semicolon preceding the ’end’ or following the ’begin’ of a block.

There are many different statements in Algol W. An example of a simple statement is the <assignment-statement>, which is employed to store a value in a declared variable. The assignment operator := is used.

An example Algol W program:

```
begin
   integer A, B, C, D;
   A := 5;
   B := -6;
   C := A + B;
   D := A - B
end.
```

The program above declares the variables A, B, C, and D to be integers, that is whole numbers, and assigns values to them.

## Order

The <declarations> part of the block is optional. A block need not have declarations. However, all blocks must contain at least one statement.

The ordering within a block is significant. All declarations must occur before any statements. It is not legal to declare a new variable in the middle of a block. It is legal, however, to begin a new block anywhere in an Algol W program where a statement may appear. Blocks can be nested within other blocks.

Example:

```
begin
   integer A, B;
   A := 5;
   B := 6;
   begin
      integer C, D;
      C := A + B;
      D := C * A
   end
end.
```

## Scope

Scope refers to the range of blocks over which a variable has meaning. A variable is only meaningful within the block in which it has been declared or defined. For example, in the program above, A and B are accessible anywhere in the program, while C and D can only be referenced in the inner block. For a more detailed explanation of scope, see "Blocks" in the section "Statements."

SIMPLE INPUT AND OUTPUT

Algol W provides several statements which allow input to be specified on data cards or in a data file.

```
Read(A)
```

stores the value given on a data card or in a data file, into the storage location designated A. Data constants can be signed constants, that is there may be a plus (+) or minus (-) sign preceding a number used as data for an Algol W program, to indicate a positive or negative value, respectively. The statement:

```
Read(A, B, C)
```

reads the first three values on a card or cards, and stores them in the corresponding locations specified in the Read statement. All 80 columns of a data punched card are read; from a file the first 256 characters of each line are read. Any characters appearing past column 256 are ignored. No data item may cross a line or card boundary. If there are only one or two values on the first data line, the next line is read, and so on until three values have been encountered. Each Read statement begins reading at column 1 of the next data line.

```
begin
   integer A, B, C;
   Read(A);
   Read(B);
   Read(C)
end.
```

The program above reads A from the first card, B from the second, and C from the third. If the data cards are as follows:

```
5   6
7
8
```

A will have the value 5, B the value 7, and C the value 8. The 6 on the first data card is ignored.

If it is desirable to continue reading from the same card, the Readon statement can be used. The sequence:

```
Read(A);
Readon(B, C)
```

reads a value from the first card, stores it in A, and then continues reading values for B and C from the same card. If there are not enough values on the current card to match the number of variables specified in the Readon statement, reading continues to subsequent cards until all necessary values have been read.

Example:

```
begin
   integer A, B, C, D;
   Read(A);
   Readon(B, C);
   Readon(D)
end.
```

If the data cards read:

```
5   -6
7   8
```

A will have the value 5, B the value -6, C the value 7, and D the value 8.

Note that column 1 of a data line is considered to be separated from column 256 of the previous line by a space.

Example:

```
begin
   integer A, B, C;
   Read(A, B, C);
end.
```

If the data lines read:

```
col. 1                                  col. 256
     |                                        |
                                              1
     2              3
     4
```

A will have the value 1, B the value 2, and C the value 3.

Algol W also provides two output statements: Write and Writeon.

```
Write(X)
```

prints the value of X. If more than one value is to be printed on one line, one can say:

```
Write(X, Y, Z)
```

If too many variables are specified for the values to fit on one line, printing continues on the next line. Each Write statement begins printing on the next line. The sequence:

```
Write(X, Y);
Write(Z)
```

causes the values of X and Y to be printed on one line, and the value of Z to be printed on the next. If it is desirable to continue writing on the same line which the previous output statement used, the Writeon statement can be employed:

```
Write(X, Y);
Writeon(Z, W)
```

prints all four values of X, Y, Z, and W on the same line.

Output statements are not confined to specifying variables. It is also possible to print a given string of characters by enclosing it in quotation marks (that is double quotes). For example:

```
Write(X, " is the value of X")
```

prints the value of X followed by the words enclosed in quotation marks, on the same line.

## Output of Real Numbers

Unless the user specifies otherwise (see "Format Variables" in the section "Basic Input and Output") real numbers (reals) are printed out with a total of seven digits and a decimal point.

Example:

```
begin
   real A, B;
   Read(A, B);
   Write(A, B)
end.
```

If the data are:

```
3.5   72.8
```

the output is:

```
3.500000    72.80000
```

When the absolute value of a real number requires more than seven digits, Algol W makes use of a representation similar to scientific notation.

Examples:

53826942.8

is represented as:

5.382694'+7

where ' (prime) is the Algol W equivalent of "times 10 to the power" in scientific notation.

0.030042183

is represented as:

3.004218'-2

See "Real" in the section "Values and Types," for more details.

## COMMENTS

The reserved word 'comment' followed by any sequence of characters, followed by a semicolon (;) is called a comment. A comment has no effect on the running of an Algol W program and is used only to improve readability of programs by including the user's explanations. Any EBCDIC character (see Appendix B) other than semicolon, including lowercase letters and others not listed in the section "Basic Symbols in Algol W," may be used in comments. This type of comment can be placed anywhere in an Algol W program that a blank would be allowed, except within a quoted string. If the terminating semicolon is omitted the following program text may be taken as part of the comment giving rise to strange errors.

A single identifier (see the section "Identifiers") inserted between the reserved word 'end' and either a reserved word, a semicolon (;), or a period (.), is also interpreted as a comment.

Comments may be written in a brief form by using the percent sign, %, to indicate both the start and the end of a comment. Comments which start with per cent may also be ended with a semicolon.

The following are all examples of comments:

```
comment  Program to calculate means;
% Add one to running total %
% Check validity ;
```

All programs should be documented with comments as an aid to others who may wish to use and understand the program and as an aid to remembering its function.

## EXAMPLE PROGRAMS

### Example Program 1

```
begin
   real A, B, C, D, E;
   Read(A, B, C);
   Readon(D, E);
   Write("Example program 1");
   Write(A, B);
   Write(C);
   Writeon(D, E)
end.
```

If the data are as follows:

```
4.2   6.3   5.7   8.5
9.5
```

the output is:

```
Example program 1
4.200000    6.300000
5.700000    8.500000    9.500000
```

### Example Program 2

```
comment
   This program reads in a sequence of real Score's, prints
   them, calculates the mean value of the Score's, and
   prints the mean value out ;

begin
   integer Num;
   real Total;
   Write("Example program 2 to calculate mean");
   comment Skip line by writing a blank,
           to make output more readable;
   Write(" ");
   Write(" ");
   Total := 0;
   comment Read in total number of Score's;
   Read(Num);
```

```
      comment Perform each statement in inner block Num times;
      for I := 1 until Num do
      begin
         real Score;
         Readon(Score);
         Writeon(Score);
         Total := Total + Score
      end;
      Write(" ");
      Write("Mean score = ", Total/Num)
   end.
```

If the data are as follows:

```
   5   85.3   91.2   46.5   56.8   99.5
```

the output is:

```
   Example program 2 to calculate mean

   85.30000    91.20000    46.50000    56.80000    99.50000

   Mean score =    75.85995
```

Example Program 3

The following program is meant to give the reader an idea of the basic structure of an Algol W program. The beginning programmer should probably postpone going through it until the concepts of arrays and procedures (subprograms) are understood.

```
   comment
      This program prints a title, reads a sequence of real
      numbers into an array Score, and prints the values of
      the numbers. It then calls the procedure Mean to
      determine the mean value. Finally, it prints the value
      of the mean score ;

   begin
      integer Num;
      Read(Num);
      Write("Example program 3 to calculate mean");
      Write(" ");
      Write(" ");

      comment
         Start new block to declare array with Num elements.
         Cannot declare array in current block since
         declarations must precede all statements, and cannot
         specify array bounds before finding value of Num. ;
```

```
        begin
           real array Score(1::Num);

           comment
              Declare function procedure Mean to calculate the
              mean. The procedure reads the array and the size
              of the array. It adds the values of the array
              Grade into the local variable Total. The final
              expression Total/N is the value of the Mean.
              Note that all declarations, including procedure
              declarations, come before statements in an Algol W
              program. ;

           real procedure Mean(real array Grade(*);
                  integer value N);
           begin
              real Total;
              Total := 0;
              for I := 1 until N do
              Total := Total + Grade(I);
              Total/N
           end Mean;

           comment Now begin the program's statements.
              Read the Score values
              into an array and print them;
           for I := 1 until Num do
           begin
              Readon(Score(I));
              Writeon(Score(I))
           end For_Loop;

           comment Print a blank line to improve readability;
           Write(" ");

           comment Invoke function procedure Mean
              and output result;
           Write("Mean score = ", Mean(Score,Num))

        end Inner_Block
      end Outer_Block.
```

If the data are as follows:

```
     5   85.3   91.2   46.5   56.8   99.5
```

the output is:

```
     Example program 3 to calculate mean

     85.30000    91.20000    46.50000   56.80000   99.50000

     Mean score =    75.85995
```

Note that the following words are interpreted as comments: "Mean" following the 'end', indicating the end of the procedure declaration Mean, "For_Loop", indicating the end of the 'for' loop, and "Inner_Block" and "Outer_Block", signifying the end of the inner and outer blocks, respectively (see the section "Comments").

# IDENTIFIERS

An identifier consists of a letter followed by zero to 254 letters, digits, and the underscore symbol (_). Thus, an identifier is between 1 and 255 characters long.

A letter is any one of the following characters:

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
```

A digit is any one of these characters:

```
0 1 2 3 4 5 6 7 8 9
```

Identifiers are names. They represent variables, arrays, procedures, record classes, record fields, labels, formal parameters and actual parameters (see the section "Values and Types"). Identifiers have no inherent meaning and can be chosen from the character set defined above. However, a reserved word cannot be used as an identifier. See Appendix D for a list of reserved words. Lower case letters are treated as their upper case equivalents when using identifiers.

Examples of legal identifiers:

```
B
B13a
Person
New_Page
```

Examples of illegal identifiers:

```
25              begins with a digit
What!           ! not part of legal set of characters
2a              begins with a digit
_X              begins with an underscore
Phone#          # not part of legal set of characters
New.page        . not part of legal set of characters
result          'result' is a reserved word
Total/Count     / not part of legal set of characters
Birth Date      blank cannot be used within identifiers
```

Every identifier used in a program must be defined. This is done in any one of the following ways:

(1) a declaration of a variable (see the section "Simple Variable Declarations") if the purpose of the identifier is to refer to a specific storage location,

(2) a declaration of a procedure (see the section "Procedures"),

(3) a label definition (see "Goto Statements and Labels" in the section "Statements") (the use of labels and goto statements, is strongly discouraged),

(4) its occurrence in a formal parameter list (see "Proper Procedures With Formal Parameters" and "Function Procedures With Formal Parameters" in the section "Procedures"),

(5) its occurrence following the reserved word 'for' in a 'for' loop, in which case it is known as a control identifier (see "Iterative Statements" in the section "Statements"), or

(6) its implicit declaration if it is a predeclared variable, procedure, or function (see Appendix E, Appendix F and Appendix G).

Examples:

(1) integer Test_Score

An integer storage location is set aside for the simple variable Test_Score.

(2)
```
L:  B := A + B;
.
.
goto L;
```

The label L is defined by its usage.

(3)
```
procedure Sort(real array Test(*); integer value Number);
```

The variables Test and Number are defined by virtue of appearing in the formal parameter list of the Sort procedure declaration.

(4)
```
for I := 1 until N do ...
```

I is the control identifier.

(5)
```
Sqrt(Y)
```

The predeclared function Sqrt computes the square root of a real number.

The recognition of the scope of a given identifier is determined by specific rules. See "Blocks," "Goto Statements and Labels," and "Iterative Statements" ('for' loops in particular) in the section "Statements" for explanations of scope rules.

# VALUES AND TYPES

An important distinction must be made between the concepts of type and value in Algol W. Every storage location used by a programmer is known as a variable. Every such storage location is of a certain type and possesses a value of that type. The type has to be declared (see the section "Simple Variable Declarations") and any value assigned to the location either has to match in type or else is converted to that type. (See "Assignment Compatibility" in the section "Arithmetic Expressions and Assignment Statements" and under "Assignment Statements" in the section "Statements".) The value of a constant is determined by the way the constant is written. The value of a variable is the one most recently assigned to it. Algol W's types of variables are divided into two classes: simple and structured.

## SIMPLE TYPES

### Integer

Integer values include zero and positive and negative numbers which do not contain a fractional part. They are expressed by an optional plus or minus sign followed by a sequence of digits, where digits are the following set of characters:

```
0 1 2 3 4 5 6 7 8 9
```

If no sign is given, the value is assumed to be positive.

Examples of integer values:

```
2
-356
0
+43
023
```

Note: Although integer values can be positive or negative, integer constants are unsigned when they occur within a program. The examples of signed integers above imply the evaluation of expressions (see the section "Constants, Variables, Expressions and Values").

Integers are stored as fullwords (in 32 bits). For a full description of their representation see Appendix J. The largest positive integer that can be stored is:

$2^{31} - 1 = 2147483647$

The smallest integer that can be stored is:

$-2^{31} = -2147483648$

The important thing to remember when using integers in Algol W is that integers are stored exactly, without any approximation. This contrasts with the storage of reals (see below) which is often only a close approximation to the actual values. As long as the value of an integer stays within the range:

$-2^{31} \leq$ <integer-value> $\leq 2^{31} - 1$

or approximately:

$-10^9 \leq$ <integer-value> $\leq 10^9$

(that is minus 1 billion and plus 1 billion), no integer overflow errors (integers outside the range) will occur. To prevent these errors, the user should convert integers which have a likelihood of becoming too big, to single-precision reals or double-precision reals by making them reals or long reals (see below).

## Real

Real values include zero and positive and negative numbers which contain a fractional part. There are two possible representations for reals in Algol W. The first is the form that is in common usage: an optional integer followed by a decimal point, followed optionally by another integer representing the fractional part. If the integer preceding the decimal point is omitted, then the integer following the decimal point must appear, and vice versa. It is illegal to have a decimal point appear alone as a real number. This form of a real value is called an unscaled real value. An example of this form is:

4156.28

The other representation, called scaled real, is similar to what is known as scientific notation. Instead of writing:

$20.3 \times 10^4$

however, in Algol W one would write:

20.3'4

The correct form is an optional integer, followed optionally by a decimal point and an integer representing the fractional part, followed by a prime ('), followed by an integer representing the power to which

10 should be raised. If the first real part (the mantissa) is omitted, it is assumed to be 1.

Examples of real values:

| | |
|---|---|
| 0.0 | zero |
| .56 | fifty-six one-hundredths |
| 42. | forty-two |
| 1.3'+6 | one point three by ten to the power six |
| -6.5 | minus six point five |
| 1.3'6 | one point three by ten to the power six |
| 4'-3 | four one-thousandths; four by ten to the minus three |
| '10 | (one times ...) ten to the power ten |

Note: Although real values can be positive or negative, real constants are unsigned when they occur within a program. The examples above of signed reals imply the evaluation of expressions.

Reals are stored as 32-bit, floating-point numbers. The first 8 bits of a real value are used to store the sign (+ or -) and an encoding of the exponent of 16 by which the number is multiplied. The last 24 bits are used to store the mantissa. For a full description of the internal representation of reals see Appendix J.

The largest possible positive real value that can be stored is 7FFFFFFF (base 16) which is equal to:

$$(1 - 16^{-6}) \times 16^{63}$$

in base 10, or approximately:

$$7.23 \times 10^{75}$$

The smallest positive real number is 00100000 (base 16) which is equal to:

$$1/16 \times 16^{-64}$$

in base 10, or approximately:

$$5.40 \times 10^{-79}$$

The corresponding range for negative reals is the same. The value 0 (base 10) can also be represented exactly as 00000000 (base 16).

All real values in an Algol W program are automatically converted to hexadecimal (that is base 16) floating-point numbers, usually with some round-off error. When real values are printed, they are converted back to decimals, possibly with an additional round-off error. Occasionally, the cumulative effect of round-off errors can be significant. If the user wants to avoid problems of this sort, it is advisable to use integers, where this is possible.

An example of a potential disaster caused by hexadecimal conversion round-off error is given in the following program:

```
begin
   real X, Y;
   X := 1.0;
   while X ¬= 2.1 do
   begin
      Y := Sqrt(X);
      Write(X, Y);
      X := X + 0.1
   end
end.
```

This program, designed to print out the values of X and the square roots of X for the range of X values starting at 1.0 and ending at 2.0 will continue forever because X will never be exactly equal to 2.1 since .1 (base 10) cannot be represented exactly with six significant hexadecimal digits. A solution to this problem is to change ¬= in the While statement to < or <=. A general rule to remember is: when comparing reals, do not use the operators = and ¬=.

In addition to the type of errors introduced by hexadecimal conversion, there also are finite precision errors due to the 32-bit limitations on reals. The precision possible is 6 significant hexadecimal digits, which is approximately equivalent to 7 significant decimal digits. If extended precision is desired, the user should convert to long real (see below).

## Long Real

Long real values are real numbers with precision extended to approximately 17 decimal places. A long real value is expressed by following the digits of a real or integer value, without a space, by the letter L. If a space is left, the L will be interpreted as an identifier.

Examples of long real values:

```
1.2L
0.2'-3L
5L
-3.1'5L
63.L
3.14159265358979L
```

Note: Although long real values can be positive or negative, long real constants are unsigned when they occur within a program. The examples of signed long reals imply the evaluation of expressions.

Long reals are stored as 64-bit, floating-point numbers. The sign of the number and an encoding of the exponent of 16 by which the number is multiplied, are stored in the first 8 bits. The remaining 56 bits are used to store the mantissa. Thus, a long real value is accurate up to 14 hexadecimal places which is approximately equivalent to 17 decimal places (more than double the precision possible for reals). The legal range and all other rules which apply to reals are the same for long reals. For more information on the internal representation of long reals see Appendix J.

## Complex

A complex value is a number composed of two numbers of type real, one representing the real part of the complex number, and the other the imaginary part. A complex number with a zero value for the real part is expressed by following a real or an integer value without a space, by the letter I. If a space is left, the I will be interpreted as an identifier.

Examples of complex values with zero real parts:

```
1I
13I
7.2I
-18.5I
5.6'3I
```

In mathematical notation, the last value would be:

$$0 + (5.6 \times 10^{3} i)$$

Note: The examples above are imaginary constants. In an Algol W program, there is no such thing as a complex constant. A complex number with a non-zero real part is expressed as a real or integer value followed by an imaginary part, connected by + or - without any embedded spaces. Thus a complex number is an expression.

Examples of complex values with non-zero real parts:

```
5-3I
-6.3+0.4I
-3.2-6.1I
```

The storage of each part of the complex number is the same as that for real values, that is two 32-bit, floating-point numbers.

## Long Complex

Long complex values are complex numbers, with precision extended to approximately 17 decimal places. A long complex value is a complex number composed of two long real numbers. A long complex value is expressed by following a complex value, without a space, by the letter L. If there is a non-zero real part of a long complex value, then both the real and the imaginary values must be followed by an L.

Examples of long complex values:

```
5IL
14.2L+16.3IL
60'3IL
.5L-'7IL
```

Note: In an Algol W program, there is no such thing as a long complex constant, just long imaginary constants. The examples above, which involve a non-zero real part, are examples of long complex expressions.

Long complex values are stored in the same way as two long reals; that is as two 64-bit, floating-point numbers.

See the section "Arithmetic Expressions and Assignment Statements," for types resulting from functions, expressions and assignment statements involving integer, real and complex values.

## Logical

There are two possible logical values: 'true' and 'false'. Both are reserved words.

Logical values are stored in single bytes (8 bits). In base 16, 'true' is represented as 01 and 'false' as 00 .

## Bits

Bits values are specified by a hash mark (#) followed by a sequence of 1 to 8 hexadecimal (base 16) digits. If fewer than 8 hexadecimal digits are specified, additional zeros are assumed between the # and the first digit. However, it is a good idea to get into the habit of specifying all 8 digits. Although the Algol W compiler supplies zero bits in this way, many other compilers do not. Bits values represent a binary succession of 0's and 1's. Each hexadecimal digit stands for 4 binary digits. A hexadecimal digit is one of the following:

```
0 1 2 3 4 5 6 7 8 9 A B C D E F
```

where the digits 0-9 correspond to the digits 0-9 base 10, and A,B,C,D,E, and F signify 10, 11, 12, 13, 14 and 15 respectively.

Examples of bit values:

| Decimal | Binary | Bits in Hexadecimal |
|---|---|---|
| 79 | 100 1111 | #0000004F |
| 14 | 1110 | #0000000E |

Bits are stored as a (fullword) linear sequence of 32 binary digits; that is a sequence of 0's and 1's.

## String

A string is any sequence of 1 to 256 EBCDIC characters enclosed by quotes (") - see Appendix B for a description of the EBCDIC character codes. A blank is considered to be a character. If a quote occurs within the sequence of characters, it must be written as two consecutive quotation marks "". The value of a string is the sequence of characters. The length of a string is the number of characters between the quotation marks.

When entering long string constants it is often convenient to split the constant over several source input records. Algol W allows string constants to be built up from a number of adjacent string constants provided that they are separated from each other by an input record boundary, or at least one blank. A string may not be defined across an input record boundary without splitting it in this way.

Examples of strings:

| String | Value | Length |
|---|---|---|
| "JOHN" | JOHN | 4 |
| """" | " | 1 |
| " " | ∅ (blank) | 1 |
| "AB""E" | AB"E | 4 |
| "52" | 52 | 2 |
| "ABC" "DEF" | "ABCDEF" | 6 |

Each character is stored as its equivalent integer encoding in 8 bits of storage - see Appendix B. Note that when numbers are used as part of a string, they are stored as characters and not as 32-bit integers or floating-point numbers.

### Reference

A reference value is a pointer to (implemented as a memory address of) a particular record occurrence of a record class. (See below for the definition of record occurrences and classes.) It may also have the value 'null', which means it does not point to any record. 'null' is a reserved word.

Reference values are stored as fullword (32-bit) integers.

## STRUCTURED TYPES

### Array

An array is an ordered set of values, all of identical type. An array can be declared to be any of the simple types described earlier: integer, real, long real, complex, long complex, bits, string, logical and reference (see the section "Arrays"). In each case, all of the elements which compose the array are of the specified type. Note that an array cannot itself be composed of elements of structured types; that is there are no arrays of arrays or arrays of records. Arrays are made up of a given number of dimensions, declared by the user. The total number of elements in an array equals the product of the number of elements in each dimension. An array element is accessed by the use of subscripts, one subscript for each dimension (see "Subscripts" in the section "Arrays").

Examples of array elements:

A(5)

means the 5th element of the one-dimensional array A.

X(4,2)

means the element in the 4th row, 2nd column of the two-dimensional array X.

Each element of an array is stored in the number of bits required for the type specified. There is at least one dimension in every array. The upper limit on the size of an array is not in terms of dimensions, but rather, in terms of the size of all the elements. The product of the number of elements in each of the first n-1 dimensions multiplied by the size of each element must be less than 32768 bytes. Arrays are stored linearly, since there is no machine structure which corresponds to the data structure of an n-dimensional array where n>1. In other words, multidimensional arrays are internally translated into one-dimensional arrays. They are stored in column-major form. Array

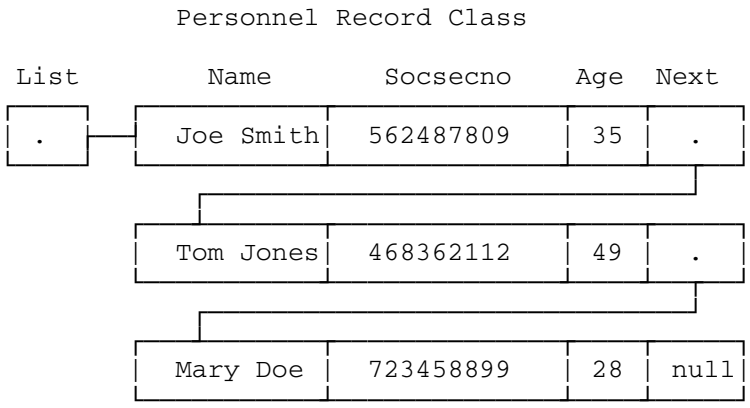
elements in multidimensional arrays are stored with their leftmost subscript changing most rapidly. For example, in a two-dimensional array, the elements of the second column are stored in succession after those in the first column, and so on until the last column. For example, an array J with 3 rows and 10 columns has its elements stored in the order:

```
J(1,1)  J(2,1)  J(3,1)  J(1,2)  ....  J(2,10)  J(3,10)
```

This mode of storage was deliberately chosen to be the same as Fortran. Note that it is the transpose of the scheme adopted by Algol 60, PL/1 and certain Pascal compilers.

## Record

A record value is an ordered set of values, in which each value may be of a different simple type. The word "record" is ambiguous in that it is often used to denote two distinct concepts: record class and record occurrence. A record class is defined as a group of variable types placed together (see the section "Records and References"). Each variable type specified is known as a field. The value of each field is of the type declared for that field. A record class defines a structure. Many record occurrences can have this same structure. The record occurrences are made up of the same fields, but have different values assigned to those fields. Since there is usually more than one record occurrence per record class, it is not enough to specify the field name of a record class in order to access a given record occurrence. Therefore, to be able to designate a particular record occurrence, one must use a variable of type reference (see above discussion of references). A reference points to a specific record occurrence, and a field within a record occurrence is accessed by specifying the field designator, that is the field name and the name of the reference variable that points to the occurrence of the record class desired.



This example contains a Personnel record class with fields Name of type string, Socsecno and Age of type integer, and Next of type reference. There are three different record occurrences belonging to the same record class. Thus, each includes the same fields but all have different values. The Next field of each record occurrence points to the next record occurrence in the list. The first record occurrence in the list is pointed to by the reference variable List. The last record occurrence signifies that there are no further records in the list by specifying a 'null' pointer in the Next field. See the section "Records and References," for declarations and assignment statements necessary for this example.

In order to access the name Joe Smith of the first record occurrence, one would say:

```
Name(List)
```

To refer to the social security number of the second record occurrence, the appropriate reference is:

```
Socsecno(Next(List))
```

since Next(List) points to the entire record occurrence following the record occurrence pointed to by the reference List, and Socsecno specifies the field designating the social security number.

The record fields are stored contiguously, each field being stored as whichever type it has been declared.

## OUTPUT OF VALUES

It should be noted that the forms of values on output depend on the current field widths for the corresponding types (see "Format Variables" in the section "Basic Input and Output") or the format string item given with the output procedure used (see the section "Format Directed Input and Output").

## SIMPLE VARIABLE DECLARATIONS

Every variable in a program must be declared before it is used. (See "For Statements" and "Goto Statements and Labels" in the section "Statements," Appendix E, Appendix F, and Appendix G for predeclared and implicitly declared variables.) The purpose of a declaration is to reserve storage space for the variable being used and to associate an identifier name with the reserved space. Declarations are only valid within a given block. They do not have any meaning outside the block in which they are declared (see the example in "Blocks" in the section "Statements").

The initial value of any declared variable is not defined. A variable receives a value only through an assignment statement or an input statement. Any value assigned to a variable of a certain type either has to match in type or else is converted to that type if possible (see "Assignment Compatibility" in the section "Arithmetic Expressions and Assignment Statements" and under "Assignment Statements" in the section "Statements").

### INTEGER, REAL, LONG REAL, COMPLEX, LONG COMPLEX AND LOGICAL

The general form of a declaration for integers, reals, long reals, complex numbers, long complex numbers and logical variables is:

```
<simple-variable-type> <identifiers>
```

where:

<simple-variable-type> can be 'integer', 'real', 'long real', 'complex', 'long complex' or 'logical'; and

<identifiers> can be a list of one or more identifiers, each separated from the next by a comma.

Examples of declarations in a partial program:

```
real Mean, Median, Mode;
integer Number;
logical Flag;
long real Average;
complex Sum;
```

These declarations set aside three storage locations for real values in Mean, Median, and Mode; one storage location Number for an integer

value; one location Average for a long real value; one location Flag for a logical value 'true' or 'false' and one location Sum for a complex value.

The semicolon (;) separates one Algol W declaration or statement from the next one.

## BITS

The declaration of a variable of type bits takes one of the following forms:

```
bits <identifiers>
bits (32) <identifiers>
```

The (32) part of the 'bits' declaration is optional since a variable of type bits is always of length 32 whether or not this is included in the declaration.

## STRING

The declaration of a variable of type string takes one of the following forms:

```
string (<integer-constant>) <identifiers>
string <identifiers>
```

where:

<integer-constant> is any constant of type integer in the range of 1 to 256 inclusive, indicating the number of characters in the string. Note that the integer following 'string' has to be a constant, not a variable. If the <integer-constant> is omitted, as in the second form, the length of the string defaults to 16.

Examples of string declarations:

```
string(20) Student_Name, Course_Name;
```

declares two strings, each with a length of 20 characters.

```
string Word;
```

declares one string with a length of 16 characters.

REFERENCE

The general form of a declaration of a variable of type reference is:

reference (<record-class-identifiers>) <identifiers>

where:

<record-class-identifiers> is a list of all the record classes to which the reference(s) can point, each separated from the next by a comma, and

<identifiers> is a list of identifiers specifying the reference variables being declared, each one separated from the next by a comma.

Examples of reference declarations:

reference(Student,Employee) List, Place;

declares two reference variables, List and Place, each of which can point to a record occurrence belonging to either of the record classes Student or Employee.

reference(Student,Course) List;

declares a reference variable List which can point to a record occurrence belonging to either of the record classes Student or Course.

reference(Student) Student_List, List;

declares two reference variables Student_List and List, both of which can point to record occurrences belonging to the record class Student.

# ARITHMETIC EXPRESSIONS AND ASSIGNMENT STATEMENTS

## EXPRESSIONS

Expressions are the means used to tell the computer to perform operations. They are made up of constants, variables, and optionally, operators such as + (addition), - (subtraction), * (multiplication) and / (division). A simple expression is either a variable or a constant.

Examples of simple expressions in Algol W:

| | |
|---|---|
| 6.2'4 | constant |
| 29 | constant |
| 4.2L | constant |
| Xnum | variable |

To form more complex expressions, simple expressions are combined with the use of unary and binary operators, and by enclosing an expression in parentheses (). Putting parentheses around an expression does not change the meaning of that expression. Unary operators are those which require only one operand, such as absolute value. Binary operators are those which involve more than one operand, such as / in Xnum/Ynum, which means the quotient of Xnum divided by Ynum.

In Algol W, the following unary arithmetic operators are used:

long

is a reserved word used to change the precision of a real expression from the default (32 bits) to long precision (64 bits).

short

is a reserved word which changes the precision of a real expression from long to short precision.

abs

is a reserved word which means the absolute value of an expression. Abs yields a value of the same type as given, for example the absolute value of a real is a real, the absolute value of an integer is an integer.

+ -

mean the same as in common arithmetic usage, that is they indicate a positive or negative value, respectively. - reverses the sign of an expression. + has no effect, but may be used for stylistic or aesthetic reasons.

Examples:

long Xnum

changes the variable Xnum from the default precision to long precision.

short(Xnum+Ynum)

changes the precision of the sum of Xnum and Ynum from long precision to short precision. Note that here, the operator + is a binary operator (see below). Both + and - can be used as unary and binary operators.

abs(Ynum)

gives the absolute value of the quantity Ynum. The equivalent mathematical formulation would be |Ynum|.

The following arithmetic binary operators are provided by Algol W:

**

is the operator used for exponentiation. Its literal translation is "raised to the power of." The expression being raised to a power can be any simple arithmetic type: integer, real, long real, complex or long complex. The power has to be of type integer.

*

means multiplication.

/

means division.

div

is also a division operator, but is used with two integer expressions and yields an integer quotient, ignoring the remainder.

rem

is an operator used with two integer expressions and yields an integer equal to the remainder of the 'div' operation. 'rem' is the equivalent of mod in modular arithmetic.

+

means addition.

-

means subtraction.

Examples:

X**2

means the square of X

Xnum ** (Num-Move)

is legal if, and only if, (Num-Move) is an integer expression.

7*Days

multiplies the current value of Days by 7.

12/7

divides 7 into 12 and yields a long real value of 1.71428571428571 (see the operator "/" table in the section "Tables of Resulting Types").

12 div 7

yields the integer quotient of 1.

12 rem 7

yields the integer remainder 5, just as in modular arithmetic, 12 mod 7 = 5.

Examples of mathematical notation and corresponding Algol W expressions:

| Mathematics | Algol W |
|---|---|
| X(Y+Z) | X*(Y+Z) |
| \|A-B+2\| | abs(A-B+2) |
| (2.4+3.6i)J | (2.4+3.6I)*J |
| 14.3x10²-7.1Zi | 14.3'2-7.1I*Z |

ASSIGNMENT STATEMENTS

Assignment statements are used to save values in storage locations. An assignment statement is of the form:

<variable> := <expression>

where:

<variable> is an identifier giving the name of a variable; and

<expression> is any constant, variable or group of these combined by unary and/or binary operators.

<variable> can also be a subscripted variable indicating an element of an array, a field designator indicating a field of a record occurrence or a substring designator indicating a portion of a string. However, these will not be discussed in this chapter. See "Assignment Statements" in the section "Statements," "Subscripts" in the section "Arrays," "Accessing Fields and Field Assignment Statements" in the section "Records and References," and "String Expressions" in the section "Strings."

The assignment operator := is translated as "is assigned the value" or "becomes." In other words, the value of the expression on the right side of := is stored in the variable referred to by the identifier on the left side. Note that it is illegal to place an expression on the left side of the assignment operator. Only a variable may appear on the left side of := .

Examples of legal assignment statements:

```
Length := abs Counter;
A := A * B;
I := I + 1;
Mean := Total/Counter;
Sum := (M + N)*Score + 5*(A - B)
```

Examples of illegal assignment statements:

```
A + B := C
```

Illegal expression on left side of :=. Left side must refer to a single storage location.

```
I = I + 1
```

Operator = is not the assignment operator.

## Multiple Assignment Statements

It is possible to combine more than one assignment into one statement. Because it is illegal for an expression to appear on the left-hand side of any assignment statement, a multiple assignment statement requires variables everywhere but on the right side of the rightmost assignment operator := . For example,

```
X := Y := B * A
```

first assigns the value of B*A to the storage location Y and then assigns the value of Y to X.

Examples of legal multiple assignment statements:

```
A := B := C := D;
B := C := (D + 4) * (H/(K + 5));
B := C := B + C
```

An example of illegal multiple assignment statement:

```
A := B+C := D
```

Illegal expression B+C on left side of assignment operator.

## PRECEDENCE

As stated above, enclosing an expression in parentheses does not change the meaning of the expression. However, it does change the meaning of an expression which includes it combined with another expression. For example:

```
A * (B + C)
```

does not mean the same thing as:

```
A * B + C
```

while:

```
(B + C)
```

is exactly equivalent to

```
B + C
```

Parentheses are used to specify the order in which expressions are to be evaluated. If parentheses are not included in an expression, the operations are performed in order of precedence. Both lists of unary

and binary operators under "Expressions" in this chapter, are listed in descending order of precedence; the operators on the higher rows are acted on before those on the lower rows. The operators on the same row have equal precedence. The combined list of arithmetic operators, in order of precedence is shown here:

```
long   short   abs
**
*   /   div   rem
+   -
```

Examples of expressions without parentheses:

```
X*Y+Z
```

is the same as (X*Y)+Z because * (multiplication) has a higher precedence than + (addition).

```
Num**3+4
```

evaluates to (Num**3) + 4 and not Num**7 because ** (exponentiation) has a higher precedence than + (addition).

When operators of equal precedence are used without parentheses, the operators are evaluated from left to right. It may be necessary to use parentheses to enforce a particular meaning when operators of equal precedence appear within an expression.

The evaluation of operators is completely specified in the description above. However, the evaluation of order of the left and right operands of a binary operator is unspecified and may occur in any order. Because of this fact, a potential problem exists when a function procedure call (see the section "Procedures") implicitly changes a variable used elsewhere in the same expression. The results of the evaluation are unpredictable, and such cases should be avoided.

## Tables of Resulting Types

The tables on the following pages give the types resulting from the operations performed on expressions of each of the possible arithmetic types:

| operator<br>+ or - | integer | real | l/real | complex | l/cmplx |
|---|---|---|---|---|---|
| integer | integer | real | l/real | complex | l/cmplx |
| real | real | real | real | complex | complex |
| l/real | l/real | real | l/real | complex | l/cmplx |
| complex | complex | complex | complex | complex | complex |
| l/cmplx | l/cmplx | complex | l/cmplx | complex | l/cmplx |

where in this and following tables:

```
l/real ........  is long real
l/cmplx .......  is long complex
```

| operator<br>* | integer | real | complex |
|---|---|---|---|
| integer | integer | long real | long complex |
| real | long real | long real | long complex |
| complex | long complex | long complex | long complex |

Long real may be substituted for real, and long complex for complex in the above table without changing the results.

| operator<br>/ | integer | real | l/real | complex | l/cmplx |
|---|---|---|---|---|---|
| integer | l/real | real | l/real | complex | l/cmplx |
| real | real | real | real | complex | complex |
| l/real | l/real | real | l/real | complex | l/cmplx |
| complex | complex | complex | complex | complex | complex |
| l/cmplx | l/cmplx | complex | l/cmplx | complex | l/cmplx |

Note the type long real which results from integer/integer.

Note that the above three tables are symmetric; i.e. the resulting type does not depend on which expression appears first.

| operator ** | integer |
|---|---|
| integer | long real |
| real | long real |
| long real | long real |
| complex | long complex |
| long complex | long complex |

It is illegal for an exponent to be of any type but integer.

| operator 'long' | result |
|---|---|
| integer | long real |
| real | long real |
| complex | long complex |

It is illegal to use the operator 'long' on an expression which is already long real or long complex.

| operator 'short' | result |
|---|---|
| long real | real |
| long complex | complex |

It is illegal to use the operator 'short' on an expression which is already of short precision, that is integer, real or complex.

It is illegal to use any type but integer expressions when performing the operations 'div' and 'rem'.

## Assignment Compatibility

When assigning the value of an expression to a variable, if the types of the expression and identifier do not match, the following conversions or errors take place:

| | | expression | | | | |
|---|---|---|---|---|---|---|
| | operator := | integer | real | l/real | complex | l/cmplx |
| variable | integer | integer | illegal | illegal | illegal | illegal |
| | real | real | real | real | illegal | illegal |
| | l/real | l/real | l/real | l/real | illegal | illegal |
| | complex | complex | complex | complex | complex | complex |
| | l/cmplx | l/cmplx | l/cmplx | l/cmplx | l/cmplx | l/cmplx |

Note that when an integer, real, or long real value is assigned to a complex variable, the value is converted to or remains a real value, and is assigned to the real part of the complex variable. When an integer, real, or long real value is assigned to a long complex variable, the value is assigned to the long real part of the long complex value. The imaginary or long imaginary part becomes zero. The values in the table are the type of the variable after the assignment has occurred. "illegal" indicates that a compiler error message would result from the attempted assignment. In general, assignment compatibility follows the principle of "widening": it is determined by whether the change in type, if any, restricts the range the value may take on. If it does, it

is illegal. (Exceptions: long real -> real and long complex -> complex)

Assume the following declarations in a partial program:

```
integer N;
real X;
long real Xx;
complex C;
```

Examples of legal assignment statements:

```
X := N;
X := 4;    (X now has the value 4.000000)
Xx := N;
X := Xx := N;
C := N;
C := X := N;
```

Examples of illegal (due to type incompatibility) assignment statements:

```
N := X;
```

Cannot assign a real value to an integer variable.

```
N := N**3;
```

The result of an integer exponentiation is long real.

```
X := N := Xx;
```

Illegal to assign a long real value to an integer variable.

When the types are different, although a legal assignment is taking place, the results are sometimes less accurate.

Assume the same declarations as above:

```
integer N;
real X;
long real Xx;
complex C;
```

then:

```
Xx := 3.1415926535897932;
```

results in Xx containing the value:

```
3.141593ggggggggg
```

where the g's are trailing garbage digits, because the L indicating long real type was omitted from the end of the real value of pi.

```
Xx := 3.1415926535897932L;
```

results in Xx containing the entire value accurate to the 17 digits listed.

```
X := 3.1415926535897932L;
```

assigns the value 3.141593 to X in accordance with the conversion process of changing a long real value to its real equivalent if assigned to a real variable.

```
Xx := X := X*N;
```

Here, X*N yields a long real value as the product of a real and an integer (see above * table). When it is assigned to X, X stores its real equivalent, that is the first seven decimal digits, and Xx takes on the equivalent long real value, that is the first seven digits followed by ten garbage digits, not the original X*N value. The last ten digits of accuracy were lost upon storage in X, a real variable.

## PREDECLARED FUNCTIONS

Algol W provides predeclared functions for the purposes of both calculation and type conversion. The form of a predeclared function call is:

<function-identifier> (<expression>)

where:

<function-identifier> is the name of the function desired; and

<expression> can be any expression as previously described, whose type is appropriate for the given function.

The function itself is treated as an expression of the type to which it belongs, meaning that it can be assigned to variables with which it is assignment compatible and can be used as an argument in proper and function procedures (see the section "Procedures").

The <expression> in a function call is known as the argument of the function. The type of the argument and the type of the resulting function value depend on the function.

The following tables list the numerical predeclared functions:

Real to Integer Conversion Functions

| Function Identifier | Function Type | Argument Type | Meaning |
|---|---|---|---|
| Truncate | integer | real | Truncated value of argument |
| Entier | integer | real | Largest integer ≤ argument |
| Round | integer | real | Rounded value of argument |
| Exponent | integer | real | Unbiased exponent used in machine representation of argument - see Appendix J |

Floating Point Conversion Functions

| Function Identifier | Function Type | Argument Type | Meaning |
|---|---|---|---|
| Roundtoreal | real | long real | Properly rounded value of argument |
| Realpart<br>Longrealpart | real<br>long real | complex<br>l/complex | Real component of argument |
| Imagpart<br>Longimagpart | real<br>long real | complex<br>l/complex | Imaginary component of argument |
| Imag<br>Longimag | complex<br>l/complex | real<br>long real | Complex equivalent of argument (uses zero as real part and argument as imaginary part) |

Roots and Powers Functions

| Function Identifier | Function Type | Argument Type | Meaning |
|---|---|---|---|
| Sqrt<br>Longsqrt | real<br>long real | real<br>long real | Positive square root of argument |
| Exp<br>Longexp | real<br>long real | real<br>long real | e (that is 2.71828..) to the power of the argument |
| Ln<br>Longln | real<br>long real | real<br>long real | Natural logarithm of argument |
| Log<br>Longlog | real<br>long real | real<br>long real | Logarithm to base 10 of argument |

Trigonometric Functions

| Function Identifier | Function Type | Argument Type | Meaning |
|---|---|---|---|
| Sin<br>Longsin | real<br>long real | real<br>long real | Sine of argument which is in radians |
| Cos<br>Longcos | real<br>long real | real<br>long real | Cosine of argument which is in radians |
| Tan<br>Longtan | real<br>long real | real<br>long real | Tangent of argument which is in radians |
| Cot<br>Longcot | real<br>long real | real<br>long real | Cotangent of argument which is in radians |

Inverse Trigonometric Functions

| Function Identifier | Function Type | Argument Type | Meaning |
|---|---|---|---|
| Arcsin<br>Longarcsin | real<br>long real | real<br>long real | Inverse sine (that is the angle in radians whose sine is the argument) |
| Arccos<br>Longarccos | real<br>long real | real<br>long real | Inverse cosine, in radians |
| Arctan<br>Longarctan | real<br>long real | real<br>long real | Inverse tangent, in radians |

The above Algol W predeclared functions return the principal values of the mathematical function in each case.

Hyperbolic Functions

| Function Identifier | Function Type | Argument Type | Meaning |
|---|---|---|---|
| Sinh<br>Longsinh | real<br>long real | real<br>long real | Hyperbolic sine of argument |
| Cosh<br>Longcosh | real<br>long real | real<br>long real | Hyperbolic cosine of argument |
| Tanh<br>Longtanh | real<br>long real | real<br>long real | Hyperbolic tangent of argument |

Special Functions

| Function Identifier | Function Type | Argument Type | Meaning |
|---|---|---|---|
| Erf<br>Longerf | real<br>long real | real<br>long real | Error function of argument |
| Erfc<br>Longerfc | real<br>long real | real<br>long real | Complementary error function of argument |
| Gamma<br>Longgamma | real<br>long real | real<br>long real | Gamma function of argument |
| Lngamma<br>Longlngamma | real<br>long real | real<br>long real | Natural logarithm of the gamma function of argument |

Complex Functions

| Function Identifier | Function Type | Argument Type | Meaning |
|---|---|---|---|
| Cxsqrt<br>Longcxsqrt | complex<br>l/complex | complex<br>l/complex | Complex square root of argument |
| Cxexp<br>Longcxexp | complex<br>l/complex | complex<br>l/complex | Complex exponential of argument |
| Cxln<br>Longcxln | complex<br>l/complex | complex<br>l/complex | Complex natural logarithm of argument |
| Cxsin<br>Longcxsin | complex<br>l/complex | complex<br>l/complex | Complex sine of argument in radians |
| Cxcos<br>Longcxcos | complex<br>l/complex | complex<br>l/complex | Complex cosine of argument in radians |

Predeclared Function Examples

Note that the table of assignment (:=) conversions and errors under "Assignment Compatibility" in this chapter also applies to functions, with the substitutions of "Function Type" for "Variable" and "Type Actually Given" for "Expression". In other words, if the wrong function type is given in a function call, either the type of the value is converted automatically, or an error message is given. The same rules apply for function calls as for assignment statements.

Assume the following declarations in a partial program:

```
integer N;
real Z, Y, X;
long real Xx;
complex C;
```

Examples of legal assignment statements:

```
Xx := Round(X) div N
```

Round(X) has type integer, and thus can be used with the 'div' operator. It is legal to assign an integer value to the long real variable Xx.

```
Z := Exp(Y*Ln(X))
```

The result Y*Ln(X) has type long real, which is converted to a real value to be a legal argument for Exp. It is legal to store the real result of the function in the real variable Z. The result of the function call is equivalent to X**Y, since X**Y=e**(Y*Ln(X)). The Exp function avoids the limitation which allows only integer exponents in expressions. In this example, Z now has the value X**Y where Y has a real value.

```
N := Truncate(Xx+X) rem Round(Xx-X)
```

The values Xx+X and Xx-X are converted to their real equivalents, Truncate and Round both result in integer values, and the result of the 'rem' operator is an integer value.

```
C := Imag(Sqrt(Cos(X)))
```

The value Sqrt(Cos(X)) is real (assuming that Cos(X) is >=0, and that Sqrt is therefore a defined function) and the Imag function takes the real value and assigns it to the imaginary part of C. It assigns a zero value to the real part of C.

```
X := Sin(N)
```

The integer N is converted to its real equivalent and the sine of N radians is stored in the real variable X.

Examples of illegal assignment statements due to assignment incompatibility:

```
N := Sin(X)
```

Sin(X) returns a real value which cannot be stored in the integer variable N.

```
N := Round(C)
```

Round requires a real or long real argument type. It does not convert a complex value to a real.

## Predeclared Function Domains of Definition

The predeclared functions described in this section are not always defined for all possible values of their arguments. A list of predeclared functions with their domains of definition and singularities, if any, is given in Appendix F.

Exceptional conditions which may arise from invalid arguments to numerical predeclared functions are described in "Predeclared Function Errors and Default Values" in the section "Miscellaneous Topics."

# CONSTANTS, VARIABLES, EXPRESSIONS AND VALUES

In Algol W, it is important to distinguish between the above four terms; constants, variables, and expressions possess values.

## CONSTANTS

The value of a constant is determined by the way in which the constant is written by the user. Constants can be categorized by type as follows:

### Arithmetic

Arithmetic constants include constants of type integer, real, long real, imaginary, and long imaginary. Algol W allows only unsigned constants to be used within a program. Therefore, a complex number is treated as an expression, being composed of an imaginary constant, a real constant, and an arithmetic operator (+ or -). It is legal, however, to have signed constants as part of the data to be read by an Algol W program.

Examples of legal constants within an Algol W program:

integer

```
5    4    120    0
```

real

```
15.2'12    4.32    '2
```

long real

```
4.2L    '5L
```

imaginary

```
3I    2.4I    1I
```

long imaginary

```
2.5IL    30'4IL
```

It is necessary to make the distinction between constants and values of the corresponding types. An arithmetic value can be positive or negative, while a constant can only have a positive value. The unary operators + and - can be interpreted as binary operators with a 0 preceding the sign. In other words, the use of a sign automatically implies the evaluation of an expression. For example:

```
-5
```

should be thought of as the expression

```
0-5
```

by the Algol W user. An error message will be returned if an expression such as

```
X*-2
```

is used in an Algol W program. The reason is that * (multiplication) has precedence over - (subtraction), and the compiler parses the expression as:

```
(X*-)2
```

which in Algol W has no meaning.

## Logical, Bits and String

Logical, bits and string constants are equivalent to logical, bits and string values (see the section "Values and Types").

## Reference

The only reference constant is 'null', indicating that the reference points to no record occurrence. The value of a reference is the record occurrence which the reference is currently pointing to.

Note that a constant is an expression, and thus may only appear on the right-hand side of the assignment operator :=.

## VARIABLES

The term variable refers to a single storage location of a simple type. Simple type means integer, real, long real, complex, long

complex, logical, bits, string and reference. The value of a variable is the value of the expression most recently assigned to it.

The following are the four possible forms of variables:

(1) <identifier> is a simple variable (see the section "Identifiers")

(2) <subscripted-variable> is an array identifier followed by a parenthesized subscript list, indicating an element of an array (see "Subscripts" in the section "Arrays")

(3) <field-designator> is an identifier followed by a parenthesized reference variable, indicating a field of a record occurrence (see "Accessing Fields and Field Assignment Statements" in the section "Records and References")

(4) <substring-designator> indicates a portion of a string (see "String Expressions" in the section "Strings")

An identifier is the name of a simple variable, a procedure, an array, or a record class.

It is legal to place a variable on either side of the assignment operator := provided the types are compatible (see "Assignment Compatibility" in the section "Statements").

## EXPRESSIONS

Expressions are the means used to tell the computer to perform operations. They are made up of constants, variables and optionally, operators. An expression can be a variable or a constant. In assignment statements, expressions appear on the right-hand side of the assignment operator :=. The left-hand side of an assignment statement must be a variable, never an expression. The following are the legal types of expressions in Algol W:

### Arithmetic Expressions

See the section "Arithmetic Expressions and Assignment Statements."

### Logical Expressions

See "Logical Expressions" in the section "Logicals."

String Expressions

See "String Expressions" in the section "Strings."

Bits Expressions

See "Simple Bits Expressions" and "Bits Expressions" in the section "Bits."

Reference Expressions

See "Creating Records" in the section "Records and References."

Functions

A function is treated as an expression of the simple type to which it has been declared - see "Function Procedures" in the section "Procedures."

Note that the value of a function can be any of the simple types of expressions, namely, arithmetic, logical, string, bits or reference.

Conditional Expressions

The two types of conditional expressions are the If expression and the Case expression, corresponding to the two types of conditional statements (see "Conditional Statements" in the section "Statements").

If Expression

An If expression takes the following form:

```
if <logical-expression> then <expression> else <expression>
```

where:

<logical-expression> consists of one or more <relations> and/or logical constants, combined by the logical binary operators 'and', 'or' or the logical unary operator ¬ (or the equivalent reserved word 'not') as defined in the section "Logicals"; and

<expression> can be any legal expression of any type.

'if', 'then', and 'else' are reserved words. If expressions must be enclosed in parentheses if they appear as parts of larger expressions. If the value of the <logical-expression> is 'true', the <expression> following the 'then' is evaluated and its value becomes the value of the conditional expression. If the value of the <logical-expression> is 'false', the expression following the 'else' is evaluated and its value becomes the value of the conditional expression.

Type of Resulting If Expression

If the <expressions> in the If expression are arithmetic, the resulting type of the entire If expression is given by the following table:

| expression type | integer | real | l/real | complex | l/cmplx |
|---|---|---|---|---|---|
| integer | integer | real | l/real | complex | l/cmplx |
| real | real | real | real | complex | complex |
| l/real | l/real | real | l/real | complex | l/cmplx |
| complex | complex | complex | complex | complex | complex |
| l/cmplx | l/cmplx | complex | l/cmplx | complex | l/cmplx |

Type is determined by the following precedence rule:

complex
real
integer

Precision is determined by the following precedence rule:

4 byte (that is short)
8 byte (that is long)

In other words, if at least one type is complex, the type of the entire If expression is complex. If at least one type has 4 byte precision, then the entire If expression has 4 byte precision. The exceptions to these rules are the long complex type resulting from integer and long complex, and the long real type resulting from integer and long real.

Note that the table is symmetric, meaning that which <expression> follows the 'then' and which <expression> follows the 'else' is

irrelevant in determining the type and precision of the entire If expression.

For all other than arithmetic types, the <expression> following 'then' and the <expression> following 'else' must match exactly in type (except for string lengths). Note that if the <expressions> are string expressions, the length of the resulting string value equals the maximum of the lengths of the string <expressions>. If necessary, blanks are appended on the right of the shorter string.

## Assignment Compatibility

If an If expression is used in an assignment statement, the type of the If expression must be assignment compatible with the variable being assigned its value. It is not enough for one of the <expressions> to be assignment compatible with the variable.

Assume the following declarations in a partial program:

```
logical L, Flag, On;
integer A, B, C;
real X;
string(7) Word;
string(5) Line;
```

Examples of assignment statements using conditional If expressions:

```
L := if Flag and ¬On then Flag else On
```

is equivalent to:

```
if Flag and ¬On
     then L := Flag
     else L := On
```

The assignment statement:

```
L := if A > B then Flag else On
```

is equivalent to:

```
if A > B
   then L := Flag
   else L := On
```

The assignment statement:

```
X := (if A > B then A else B)+(if X < Y then X else Y)
```

is equivalent to:

```
if (A > B) and (X < Y)
   then
      X := A + X
   else
   if ¬(A > B) and ¬(X < Y)
      then
         X := B + Y
      else
      if ¬(A > B) and (X < Y)
         then X := B + X
         else X := A + Y
```

The assignment statement:

```
Word := if Line(1|1)="A" then Line else "A"
```

is equivalent to:

```
if Line(1|1)="A"
   then Word := Line
   else Word := "A"
```

An example of an illegal assignment statement:

```
A := if L then B else X
```

The type of the If expression is real, and it is illegal to assign a real expression to an integer variable.

## Case Expressions

A Case expression takes the following form:

```
case <integer-expression> of (<expressions>)
```

where:

<integer-expression> is any expression whose value is of type integer; and

<expressions> is a list of legal expressions of any type, each one separated from the next by a comma, in accordance with the table and precedence rules given for If expressions above. In other words, if at least one of the <expressions> in the Case expression is of complex type, the resulting Case expression is of complex type. If one of the <expressions> is 4 bytes (that is short precision), the resulting Case expression has short precision.

'case' and 'of' are reserved words.

(Note the exceptions in the long complex type resulting from a cross between integer and long complex, and the long real type resulting from integer and long real.)

If the <expressions> in the Case expression are of other than arithmetic type, they must match exactly (except for string lengths). Note that if the <expressions> are string expressions, the length of the resulting string value equals the maximum length of the string expressions. If necessary, blanks are appended on the right of the shortest string.

The assignment compatibility rule for If expressions applies to Case expressions as well.

Just as in a Case statement (see "Conditional Statements" in the section "Statements), the value of the <integer-expression> must be greater than 0 and less than or equal to the number of expressions listed after 'of'. When a Case expression is used, the expression evaluated is the one whose position in the <expressions> matches the value of the <integer-expression>. The value of that expression becomes the value of the entire Case expression. A Case expression must be enclosed in parentheses if it appears as part of a larger expression.

Assume the following declarations in a partial program:

```
string(15) Cards;
integer Suit;
```

Example of conditional Case expression in an assignment statement:

```
Cards := case Suit of
   ("SPADES","HEARTS","DIAMONDS","CLUBS");
```

is equivalent to:

```
case Suit of
begin
   Cards := "SPADES";
   Cards := "HEARTS";
   Cards := "DIAMONDS";
   Cards := "CLUBS"
end
```

Note that the value of a conditional expression can be any of the simple types of expressions, namely, arithmetic, logical, string, bits or reference.

## Block Expressions

A block expression is a type of block (see "Blocks" in the section "Statements") composed of a 'begin', followed optionally by declarations, followed by a series of statements, followed by an expression of any type, followed by an 'end'. There must not be a semicolon following the expression. The value of the expression becomes the value of the block expression.

Example:

```
begin
   integer Num;
   Read(Num);
   Write(Num);
   Num ¬= 0
end
```

is a logical block expression whose value equals 'true' if Num has any value other than 0, and whose value equals 'false' if Num equals 0 .

A block expression can be of any simple type, and can be inserted anywhere in an Algol W program where it is legal to use an expression. Examples of good uses for block expressions are given in the explanation of While statement loops (see "Iterative Statements" in the section "Statements") and in "Function Procedures" in the section "Procedures."

Note that the value of a block expression can be any of the simple types of expressions, namely, arithmetic, logical, string, bits or reference.

# ARRAYS

An array is a data structure composed of a collection of elements that are all of the same simple type. An array with one dimension is a vector, that is a linear collection of values. An array with two dimensions corresponds to a rectangular matrix. Arrays with three or more dimensions correspond to rectangular solids of the given dimension.

An array must be declared as any one of the simple types: integer, real, long real, complex, long complex, logical, string, bits or reference. An array cannot be composed of structured types, that is there are no arrays of arrays or arrays of records.

## ARRAY DECLARATIONS

### Integer, Real, Long Real, Complex, Long Complex and Logical Array Declarations

The form of an array declaration for arrays of integers, reals, long reals, complex, long complex or logicals is:

<simple-variable-type> array <identifiers> (<bound-pairs>)

where:

<simple-variable-type> can be 'integer', 'real', 'long real', 'complex', 'long complex' or 'logical';

<identifiers> is a list of one or more identifiers, each separated from the next by a comma, each being the name of an array being declared;

<bound-pairs> is a list of one or more bound pairs, each separated from the next by a comma.

The form of a <bound-pair> is:

<integer-expression-1> :: <integer-expression-2>

where:

<integer-expression> can be any combination of constants, variables, and/or operators, resulting in a legal expression of type integer;

<integer-expression-1> indicates the lower bound of the array dimension;

<integer-expression-2> indicates the upper bound of the array dimension.

Note that <integer-expression-1> may be greater than <integer-expression-2>, in which case an array with no elements is being declared.

The total number of bound pairs listed is the number of dimensions in the array.

Examples:

```
integer array Number, Times(1::50)
```

declares two, one-dimensional arrays of 50 elements of type integer.

```
real array Score(1::N,K::M+N)
```

declares a two-dimensional array of N*(M+N-K+1) elements, each of type real. Note that the values N, K, and M+N must evaluate to integer types to be legal expressions for bound-pairs.

```
logical array Switch(1::N,1::N,1::N)
```

declares a three-dimensional array containing N cubed logical elements, each of which can contain the value 'true' or 'false'.

## String Array Declarations

The form of a string array declaration is either of the following:

```
string(<integer-constant>) array <identifiers> (<bound-pairs>)

string array <identifiers> (<bound-pairs>)
```

where:

<integer-constant> is any constant of type integer in the range of 1 to 256 inclusive, indicating the number of characters in each element of the array(s) declared;

<identifiers> and <bound-pairs> are as defined above.

If the <integer-constant> is omitted, as in the second type of string array declaration shown above, the lengths of the strings default to 16.

Examples:

```
string(15) array Word(1::Size)
```

declares a one-dimensional array Word with Size number of elements, each element being a 15-character string.

```
string array Street(I::K,M::N)
```

declares a two-dimensional array Street with (K-I+1)*(N-M+1) number of elements, each element being a 16-character string.

## Bits Array Declarations

The form of an array declaration for arrays of type bits is either of the following:

```
bits array <identifiers> (<bound-pairs>)
bits(32) array <identifiers> (<bound-pairs>)
```

where:

<identifiers> and <bound-pairs> are as defined above.

The (32) part of the 'bits' declaration is optional in that a variable of type bits is always of length 32 whether or not this is included in the declaration.

Example:

```
bits array Check(-2::2)
```

declares a one-dimensional array of five elements, each element containing a value of type bits.

## Reference Array Declarations

The form of an array declaration for arrays of type reference is:

```
reference (<record-class-identifiers>) array
    <identifiers> (<bound-pairs>)
```

where:

<record-class-identifiers> is a list of all the record classes (previously declared), which the reference can point to, each one separated from the next by a comma;

<identifiers> is a list of identifiers specifying the reference variables being declared, each one separated from the next by a comma;

<bound-pairs> are as defined under "Array Declarations" earlier in this chapter.

Example:

```
reference(Student,Course) array Number(1::10)
```

declares a one-dimensional array of ten elements, each element a reference variable, which can point to any record occurrence belonging either to the Student record class or to the Course record class.

Note that having arrays of references is functionally equivalent to arrays of records.

## SUBSCRIPTS

The total number of elements in an array is the product of the elements in each dimension. The total number of elements in each dimension equals the upper bound - the lower bound + 1. If the upper bound is less than the lower bound, the array being declared has zero elements in it. A given element is referred to with the use of a subscript list, one subscript for each dimension. A subscript must be an integer expression. A subscript list must contain one or more subscripts. The number of subscripts needed to specify an element of an array equals the number of dimensions in the array. Each subscript in an array element specification is separated from the next by a comma. Each element in an array has the same properties of a simple variable of the array type. In order to use an array element as a variable, the array identifier is followed by a parenthesized subscript list. Each individual element of the array is called a <subscripted-variable>.

Example:

```
complex array C(-3::2,-2::0)
```

declares an array of 6x3=18 complex-valued elements.

The elements of the above array are:

```
C(-3,-2)     C(-3,-1)     C(-3,0)
C(-2,-2)     C(-2,-1)     C(-2,0)
C(-1,-2)     C(-1,-1)     C(-1,0)
C(0,-2)      C(0,-1)      C(0,0)
C(1,-2)      C(1,-1)      C(1,0)
C(2,-2)      C(2,-1)      C(2,0)
```

## DYNAMIC ALLOCATION

An array has to be declared before it is used. However, since the bound-pairs are made up of integer expressions, rather than constants, and because it is legal to begin a new block anywhere in an Algol W program, an array can be a different size each time the block in which the array is declared is entered. Storage for arrays is allocated at execution time, not at compile time. This is called dynamic storage allocation. The number of dimensions in an array is static, however.

Example of a partial program:

```
integer N;
for I := 1 until 10 do
begin
   Read(N);
   begin
      real array Score(1::N);
      ...;
      N := N + 1
   end
end;
```

Each of the ten times the outer block is entered, a new value for N is read. The inner block defines a one-dimensional array Score of real values, with a different number N of elements each time the inner block is entered. The size of the array is not changed by incrementing N in the inner block.

## ARRAY ASSIGNMENTS

There is no such thing as an array assignment statement. In other words, it is not possible to assign a value to an entire array. Each element of an array must be individually assigned a value.

Example of array assignments in a partial program:

```
begin
   string(20) array Subject(1::25);
   string(20) array Object(1::25);
   integer array Subjlen(1::25);
   integer array Objlen(1::25);
   for  J := 1 until 25 do
   begin
      Subject(J) := " ";
      Object(J) := " ";
      Subjlen(J) := 0;
      Objlen(J) := 0
   end;
    .
    .
end.
```

The above program declares four arrays, and initializes each string array element to blanks and each integer array element to 0. It would not have been syntactically legal to write:

```
Subject := " ";
Object := " ";
Subjlen := 0;
Objlen := 0
```

as Algol W allows assignments to only one array element at a time.

## SAMPLE PROGRAMS

### Array Sample Program One

The following program makes use of dynamic allocation of arrays to find the largest and smallest values of a list of N scores. The first data card gives the number of scores. Each of the subsequent cards contains a score value.

```
     begin
        integer N;
        Read(N);
        begin
           comment Note the use of dynamic allocation
                   in the following statement;
           real array Score(1::N);
           real Big, Small;
           Read(Score(1));
           Big := Small := Score(1);
           for I:= 2 until N do
           begin
              real Grade;
              Read(Score(I));
              Grade := Score(I);
              if Grade > Big then Big := Grade
                 else if Grade < Small then Small := Grade
           end;
           Write(Big, Small)
        end
     end.
```

If the data are as follows:

```
     8
     20.2
     95.4
     89.3
     75.6
     18.5
     56.3
     98.2
     87.1
```

the output from this program is:

```
     98.20000   18.50000
```

Array Sample Program Two

The following sort program sorts an array Score of N elements from high to low, by keeping track of subscripts. The scheme is to compare all the elements to the first one, determine which is the largest, and then if the largest is not already the first element, to have the largest change places with the first. Next, the second element is compared with the following elements and the same pattern is repeated until the N-1th element is compared with the Nth, and the whole array is ordered from high to low. The inner 'for' loop keeps track of the subscript of the largest element in the unordered part of the array. The outer 'for' loop moves down the array as the previous elements have been sorted, and makes exchanges after each set of comparisons has been made.

Large is initialized to the subscript of the first unordered element, and is replaced by the subscript of any larger element. First_Unor_Subs is the subscript of the first element of the unordered part of the array. J is the subscript of the array element being compared to the largest element. I is the variable which controls how many passes the sorter should make.

```
begin
   integer N;
   Read(N);
   begin
      integer array Score(1::N);
      for I:= 1 until N do
         Read(Score(I));
      for I := 2 until N do
      begin
         integer Large, First_Unor_Subs;
         Large := First_Unor_Subs := I - 1;
         for J := I until N do
         if Score(Large) < Score(J)
            then Large := J;
         if Large ¬= First_Unor_Subs then
         begin
            integer Temp;
            Temp := Score(First_Unor_Subs);
            Score(First_Unor_Subs) := Score(Large);
            Score(Large) := Temp
         end
      end;
      for I := 1 until N do Write(Score(I))
   end
end.
```

If the data are as follows:

```
5
25
11
49
81
63
```

the program goes through the following steps:

| | I | J | Large | First_Unor_Subs | Score |
|---|---|---|---|---|---|
| I=2 | | | | | |
| pass | 2 | 2 | 1 | 1 | [25 11 49 81 63] |
| | 2 | 3 | 1 | 1 | [25 11 49 81 63] |
| | 2 | 4 | 3 | 1 | [25 11 49 81 63] |
| | 2 | 5 | 4 | 1 | [25 11 49 81 63] |
| I=3 | | | | | |
| pass | | | | | [81 11 49 25 63] |
| | 3 | 3 | 2 | 2 | [81 11 49 25 63] |
| | 3 | 4 | 3 | 2 | [81 11 49 25 63] |
| | 3 | 5 | 3 | 2 | [81 11 49 25 63] |
| I=4 | | | | | |
| pass | | | | | [81 63 49 25 11] |
| | 4 | 4 | 3 | 3 | [81 63 49 25 11] |
| | 4 | 5 | 3 | 3 | [81 63 49 25 11] |
| I=5 | | | | | |
| pass | | | | | [81 63 49 25 11] |
| | 5 | 5 | 4 | 4 | [81 63 49 25 11] |

Note that the values given are those defined at the beginnings of passes through the inner loop. Note also that the array is already sorted at the beginning of pass 4, but the program continues through pass 5. The output from this program is:

```
81
63
49
25
11
```

# LOGICALS

## DECLARATIONS

Logical variables can be declared as such, using the following form:

```
logical <identifiers>;
```

where:

<identifiers> is a list of identifiers, each separated from the next by a comma, each representing a logical variable.

Examples:

```
logical Switch;
logical On, Flag;
```

The only two possible values which can be assigned to a logical variable are 'true' and 'false', both reserved words.

## RELATIONS

A relation is a comparison between two expressions. It can have either of the two logical values: 'true' or 'false'. Its form is one of the following:

```
<expression-1> <relational-operator> <expression-2>
<reference-variable> is <record-class-identifier>
```

where:

<relational-operator> can belong to one of the following two groups:

The equality operators:

| | |
|---|---|
| = | equal to |
| ¬= | not equal to ('not =' is an alternative) |

The inequality operators:

```
<      less than
<=     less than or equal to
>      greater than
>=     greater than or equal to
```

The equality operators test the equality or inequality of the values of two expressions. If an equality operator is used, then expressions of any arithmetic type can replace <expression-1> and <expression-2>. If <expression-1> is of type logical, bits, string or reference, then <expression-2> must be of type logical, bits, string or reference, respectively. In other words, it is possible to test the equivalence of two arithmetic values, two logical values, two bits values, two string values, or two reference values. It is not possible, however, to compare an arithmetic expression with a string expression. For example, 1="ONE" is not a valid relation. In general, it is illegal to test the equivalence of two expressions whose types do not match. (In this context, an "arithmetic" type may be assumed to include integer, real, long real, complex and long complex.) If an inequality operator is used, only types of expressions whose values imply ordering can be compared. It is legal to compare integer, real, and long real expressions to any integer, real, and long real expressions. It is also legal to compare two string expressions. The meaning of an inequality comparison between strings is the lexical comparison. A string is less than another string if it occurs first in the dictionary. Any EBCDIC character is less than another EBCDIC character if its integer encoding is less than the integer encoding of the other character (see "String Comparisons" in the section "Strings" and Appendix B). It is illegal, however, to compare two logicals, two bit values, two complex values, or two reference values in an inequality relation, because these values are not considered ordered values. Also, just as with the equality operator, it is illegal to compare expressions whose types do not match. (In this context, an "arithmetic" type may be assumed to include integer, real and long real.)

<reference-variable> is an identifier, a subscripted variable or a field designator of type reference;

<record-class-identifier> is an identifier previously declared a record class;

'is' tests whether the reference variable specified on the left of 'is' is currently pointing to a record occurrence belonging to the record class specified on the right side of 'is' (see the section "Records and References").

Assume the following declarations in a partial program:

```
integer I, J;
real X;
complex Com, Plex;
string(20) Coursename;
record Course(string(20) Name; integer Number);
comment See the section "Records and References"
        for record class declarations;
reference(Student,Course) List1, List2;
logical L, Ll;
```

Examples of legal relations:

```
2*X <= 4
```

```
Com = Plex
```

```
L ¬= Ll
```

```
Com ¬= I
```

```
Coursename <= "ALGEBRA"
```

Note that when string comparisons are made, if the strings are of unequal length, the shorter one is extended to the length of the longer by padding it with blanks on the right (for the comparison only).

```
Number(List1) = X
```

Here, an integer field of a record occurrence is being compared to a real.

```
Name(List1) ¬= Name(List2)
```

```
List1 is Student
```

has the value 'true' if and only if List1 is currently pointing to a record occurrence belonging to class Student. It is 'false' if it is pointing to a record occurrence of class Course or if it contains the value 'null'.

```
List1 = List2
```

Here, two reference variables are compared. The value is 'true' only if both are pointing to the same record occurrence or if both have the value 'null'. (Pointing to the same record class is not sufficient for a 'true' value here.)

Examples of illegal relations:

```
Com <= I
```

Cannot use an inequality operator to compare expressions of any types but string, integer, real or long real.

```
Com > Plex
```

Cannot use an inequality operator to compare expressions of any types but string, integer, real or long real.

```
List1 is List2
```

Operator 'is' must be followed by a record class identifier, not a reference identifier.

```
L = I + J
```

Cannot compare a logical expression with an integer expression.

```
List1 > List2
```

Cannot use an inequality operator to compare expressions of any types but string, integer, real or long real.

## LOGICAL EXPRESSIONS

A logical expression consists of one or more relations, logical variables, and/or logical constants 'true' and 'false', combined by the logical binary operators 'and', 'or' or the logical unary operator ¬ or 'not' (both equivalent and signifying not).

The value of a logical expression formed by using 'and' is 'true' if, and only if, the values of both operands are 'true'. The value of a logical expression formed by using 'or' is 'false' if, and only if, the values of both operands are 'false'. The operator ¬ or 'not' changes the value of a 'true' expression to 'false' and a 'false' expression to 'true'. The following chart illustrates these properties:

| P | Q | ¬P | ¬Q | P and Q | P or Q |
|---|---|---|---|---|---|
| true | true | false | false | true | true |
| false | true | true | false | false | true |
| true | false | false | true | false | true |
| false | false | true | true | false | false |

Examples of logical expressions:

```
L and (I <= J) or ¬Ll

(X + Y) = I

List1 ¬= null

L or Ll and P

(X + Y < I) or (L ¬= Ll) and (I > 5)

¬(L and Ll) or (X <= Y)

P
```

Note that although it is syntactically correct, there is never any good reason to compare a logical expression with a logical constant.

Examples:

```
P
```

has the value 'true' if P is 'true' and the value 'false' otherwise. An equivalent but unnecessary construct would be:

```
P=true
```

P is the preferred version.

```
¬P
```

has the value 'true' if P is 'false' and the value 'true' otherwise. An equivalent but unnecessary construct would be:

```
P=false
```

¬P is the preferred version.

Note that there is a potential problem caused by logical variables being initialized by Algol W to a value which is taken as neither 'true' nor 'false'.

Example:

```
begin
   logical A;
   while ¬A do
   Write("Loop")
end.
```

The above program will loop infinitely, because A is evaluates to 'false', and so ¬A evaluates to 'true'. This may change at some future

date without warning. NO program should EVER depend upon the values of uninitialized variables.

## PRECEDENCE

If no parentheses are used to enclose relations within a logical expression, the expression is evaluated according to the rules of precedence. The following list gives the order of precedence for arithmetic, relational, and logical operators in descending order, that is the operators on the higher rows are acted on before those on the lower rows. The operators on the same rows have equal precedence.

```
abs long short
**
*   /   div   rem
+   -
<   <=   >   >=   =   ¬=   is
¬
and
or
```

Example:

```
L1 or ¬L and L11
```

is equivalent to:

```
L1 or ((¬L) and L11)
```

If no parentheses are used to enclose relations within a logical expression containing operators of equal precedence, the expression is evaluated from left to right. As much of the expression as is necessary to determine the final result will be evaluated. Some elements of the expression may therefore not be evaluated.

## LOGICAL ASSIGNMENT STATEMENTS

A logical assignment statement takes the following form:

```
<logical-variable> := <logical-expression>
```

where:

<logical-variable> is an identifier, a subscripted variable or a field designator of type logical; and

<logical-expression> is as defined previously in this chapter.

Multiple logical assignment statements are legal and are of the same form as arithmetic multiple assignment statements. In other words, there must be a logical variable, not an expression on the left side of the assignment operator := .

Logical values may only be assigned to variables of type logical.

Assume these declarations in a partial program:

```
logical Flag, Switch, L;
integer I, J, K;
record Student(string(20) Name);
record Course(integer Num);
comment See the section "Records and References"
        for declarations of record classes;
reference(Student,Course) List1, List2;
```

Examples of legal assignment statements:

```
Switch := true
L := (I < J) or (J < K)
L := (I < J) and (J < K)
Flag := List1 is Student
Switch := Flag := L
Flag := Switch := false
```

Examples of illegal assignment statements:

```
Switch := true := L
```

Illegal to have an expression rather than a variable on the left side of :=.

```
Switch := Flag or L := L
```

Illegal to have an expression rather than a variable on the left side of := .

PREDECLARED FUNCTIONS

Algol W provides a predeclared function of type logical.

| Function Identifier | Function Type | Argument Type | Meaning |
|---|---|---|---|
| Odd | logical | integer | Returns 'true' if argument is odd integer, and 'false' if argument is even. |

Example:

```
Odd(1)
```

has the value 'true'.

# STRINGS

## STRING DECLARATIONS

The declaration of a variable of type string takes either of the following forms:

```
string (<integer-constant>) <identifiers>
string <identifiers>
```

where:

<integer-constant> is any constant of type integer, in the range of 1 to 256 inclusive, indicating the number of characters in the string;

<identifiers> are one or more identifiers, each separated from the next by a comma, each referring to one string variable.

Note that the integer following 'string' must be a constant, not a variable. If the <integer-constant> is omitted, as in the second form shown above, the length of the string defaults to 16.

Examples of string declarations:

```
string(20) Studentname, Coursename;
```

declares two strings, each with a length of 20 characters.

```
string(25) array State(1::50);
```

declares an array of 50 elements, each of which is a 25-character string.

## STRING EXPRESSIONS

A string expression designates part or all of a given string. It is expressed in one of the following forms:

```
"<string-constant>"
<string-variable>
<string-variable> (<integer-expression>|<integer-constant>)
```

where:

<string-constant> is any sequence of up to 256 EBCDIC characters, listed in Appendix B, and

<string-variable> is an identifier, a subscripted variable or a field designator of type string.

Each character in a string has a position number, starting with position 0 for the leftmost character and ending with position n-1 for a string of n characters.

<integer-expression> is any expression which evaluates to a value of type integer, and specifies the beginning position within the string being designated; and

<integer-constant> is a constant of type integer, with positive value, which specifies the length of the substring designated.

The sum of the <integer-expression> and the <integer-constant> must be no greater than the total length of the string. Note that the <integer-constant> must be a constant, not a variable.

The third form of a <string-expression> is known as a <substring-designator>. Both the <string-variable> and the <substring-designator> are simple variables of type string. They may be used wherever it is legal to use a string variable in an Algol W program, for example on the left-hand side of the assignment operator := in a string assignment statement.

The double slash symbol (//) may be used as an alternative to the vertical bar (|) in a substring designator.

Assume the following declarations in a partial program:

```
string(8) Word;
string(10) array State(1::50);
integer N, J;
real X, Y;
```

Examples of legal string expressions:

Word

refers to entire string Word.

Word(0|8)

also refers to entire string Word.

State(2)(3//6)

refers to the 4th to 9th characters inclusive, of the second string in array State.

Word(N+J|4)

refers to the (N+J+1)th to (N+J+4)th characters in the string Word.

Examples of illegal string expressions:

Word(1|0)

Illegal to have length = 0.

Word(N|J)

Illegal to have variable length.

State(J)(X|4)

Illegal to have real position. Position must be integer.

Word(1|8)

Illegal for sum of position + length to be greater than total length of Word.

## STRING COMPARISONS

String comparisons allow strings to be alphabetized as well as checked for specific values. A string comparison is one example of the general relations discussed in the section "Logicals." It is of the form:

<string-expression> <relational-operator> <string-expression>

where:

<string-expression> is as defined previously; and

<relational-operator> can be any one of

= ¬= < <= > >=

The symbol 'not =' may be used in place of '¬='.

Every EBCDIC character has an integer encoding associated with it (see Appendix B). Thus, a given string character value is said to be

less than another character value if its integer encoding is less than the integer encoding of the other character. If two strings being compared are not of equal lengths, the shorter one is extended to the size of the longer one by appending blanks on the right (for the comparison only). Comparison of strings involves a comparison of each character in the string. Therefore, two strings will have equal values if and only if the corresponding characters in each string are identical.

Examples:

| Comparison | Value |
|---|---|
| "C" < "D" | true |
| "AB"<"AB " | false |
| "*" > "F" | false |
| "2" >="4" | false |
| "01" = "1" | false |
| "XY"="XY  " | true |

If Word is a string of length 9 with the value "ALGEBRAIC", then

| | |
|---|---|
| Word(0\|4)="ALGE" | true |
| Word="ALGEBRAIC" | true |
| Word(5\|1)="R" | true |

## STRING ASSIGNMENT STATEMENTS

A string assignment statement takes either of the following forms:

```
<string-variable> := <string-expression>
<substring-designator> := <string-expression>
```

where:

<string-variable> is an identifier, a subscripted variable, or a field designator of type string; and

<string-expression> and <substring-designator> are as defined previously.

The length of the string on the right side of the assignment operator := must be less than or equal to the length of the string on the left side. If the length of the string on the right side is shorter than that of the string being assigned its value, the rightmost part of the left string is padded with blanks. If the string on the left side is a substring designator, its rightmost part, out to the length of the designated substring, is padded with blanks. The rest of the left string is unchanged; if it has not been assigned a value yet, it remains undefined.

Multiple string assignment statements are legal and are of the same form as multiple arithmetic assignment statements. String constants and expressions cannot be on the left side of the assignment operator :=. Only string variables and substring designators may appear on the left side of the assignment operator.

Assume the following declarations in a partial program:

```
string(10) A;
string(5) Word;
string(4) Line;
```

Examples of legal string assignment statements in a program:

```
Word := "WHEN";
```

Assigns the first four characters of Word the values W, H, E and N respectively. The fifth character is assigned a blank.

```
Word(0|3) := Word(1|3);
```

Shifts 3 letters in Word to the left. Word now has the value HENN followed by a blank.

```
Line := Word(0|4);
```

assigns Line the value HENN.

```
Line := "TT";
```

assigns the characters TT to the first two positions in LINE. The other two positions are padded with blanks. They do not retain their previous values NN.

```
Word := Line := "COMP";
```

assigns the characters COMP to Line and to Word. Word has a blank in its fifth position.

```
A(0|5) := "12345";
```

assigns the characters 12345 to the first five positions in A. The relation A="12345" has a value of 'false' because the last five positions of A are undefined, not blank.

```
A(0|5) := "1234";
```

assigns the characters 1234 to the first four positions in A. The fifth position is a blank and positions six through ten remain undefined.

Assume the following declarations and statements in a partial program:

```
string(5) Word;
string(4) Line;
Word := "THERE";
```

Examples of illegal assignment statements:

```
Line := "ABCDE";
```

```
Line := Word;
```

Illegal for string on left side of assignment operator := to be shorter than string on right side.

```
Line := "HERE" := Word(0|4);
```

Illegal for string expression to be on left side of :=.

## PREDECLARED FUNCTIONS

Algol W provides several predeclared functions dealing with strings. In the following table, the characters in the prototype formats are:

| | |
|---|---|
| D | decimal digit in a mantissa or integer |
| E | decimal digit in an exponent |
| A | hexadecimal digit in a mantissa or integer |
| B | hexadecimal digit in an exponent |
| + | sign (blank for positive mantissa or integer) |
| ∅ | blank |

Each exponent is unbiased. Decimal exponents represent powers of 10; hexadecimal exponents represent powers of 16. Each mantissa (except 0) represents a normalized fraction less than 1. Leading 0's are not suppressed.

The internal representation of numeric values on System/370 type machines is fully discussed in Appendix J.

| Function Identifier | Function Type | Argument Type | Meaning |
|---|---|---|---|
| Decode | integer | string(1) | Integer encoding of character (see Appendix B) |
| Code | string(1) | integer | The character corresponding to the absolute value of the (Argument REM 256) (this is the inverse of the Decode function) |
| Base10 | string(12) | real | String encoding of argument with format ⌀+EE+DDDDDDD |
| Longbase10 | string(20) | long real | String encoding of argument with format ⌀+EE+DDDDDDDDDDDDDD |
| Base16 | string(12) | real | String encoding of argument with format ⌀⌀+BB+AAAAAA |
| Longbase16 | string(20) | long real | String encoding of argument with format ⌀⌀+BB+AAAAAAAAAAAAAA |
| Intbase10 | string(12) | integer | String encoding of argument with format ⌀+DDDDDDDDDD |
| Intbase16 | string(12) | integer | Unsigned, two's complement string encoding of argument with format ⌀⌀⌀⌀AAAAAAAA |
| Date | (see the description under "Clock Functions" in the section "Miscellaneous Topics") | | |

Examples:

```
Decode("8") - 240
```

   has the integer value 8.

```
Code(230)
```

   has the string value "W".

```
Intbase10(21)
   has the string value " 0000000021".
Intbase16(21)
   has the string value " 00000015".
Intbase10(-21)
   has the string value " -0000000021".
Intbase16(-21)
   has the string value " FFFFFFEB".
Base10(5.3'8)
   has the string value " +09 5300001".
Base16(1.0)
   has the string value " +01 100000".
```

# BITS

## BITS DECLARATIONS

The declaration of a variable of type bits takes either of the following forms:

```
bits <identifiers>
bits (32) <identifiers>
```

where:

<identifiers> is a list of one or more identifiers, each separated by a comma, each representing a bits variable.

The (32) part of the 'bits' declaration is optional in that a variable of type bits is always of length 32, whether or not this is included in the declaration.

## CONSTANTS

Bits constants are specified by a hash mark (#), followed by a sequence of 1 to 8 hexadecimal digits, representing 32 binary digits (see "Bits" in the section, "Values and Types").

## SIMPLE BITS EXPRESSIONS

A <simple-bits-expression> consists of one or more bits constants and/or bits variables, combined by the logical operators 'and', 'or', ¬ or 'not' (both meaning not). A bits variable is referred to by an identifier, a subscripted variable or a field designator of type bits.

Examples of <simple-bits-expressions>:

```
A and ¬B
(X or Y) and (C or ¬D)
A or #0000005A
```

The operator ¬ or 'not' changes all 0's in a bits sequence to 1's and vice versa. The 0 in a simple bits expression corresponds to 'false' in a logical expression, and the 1 corresponds to 'true'. The value of

each bit of a simple bits expression formed by using 'and' is 1 if, and only if, both bits in corresponding positions are 1. The value of each bit of a simple bits expression formed by using 'or' is 0 if, and only if, both bits in corresponding positions are 0. The following chart illustrates this:

| X | Y | ¬X | X and Y | X or Y |
|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 1 | 0 | 1 | 1 |

BITS EXPRESSIONS

A bits expression takes one of the following forms:

<simple-bits-expression> <shift-operator> <integer-expression>
<simple-bits-expression>

where:

<simple-bits-expression> is as defined above;

<shift-operator> is either 'shl', meaning shift left, or 'shr', meaning shift right;

<integer-expression> can be any arithmetic expression of type integer.

The absolute value of the integer expression gives the number of positions the bits value is to be shifted. Vacated bit positions are assigned the value 0.

Assume the following declarations and assignments in a partial program:

```
bits A, B;
integer I;
A := #00000002;
B := #0000001E;
I := 16;
```

Examples of bits expressions:

```
A or (B shl 2)

   has the value #0000007A

B or A

   has the value #0000001E

¬(B or A)

   has the value #FFFFFFE1

(¬(B or A)) shr 7

   has the value #01FFFFFF

A and ((¬(B or A)) shr 7)

   has the value #00000002

A shl (-8)

   has the value #00000200

B shl (2*I)

   has the value #00000000
```

PRECEDENCE

If no parentheses are used to enclose simple bits expressions within a bits expression, the expression is evaluated according to the rules of precedence. The following lists the order of precedence for arithmetic, relational, logical, and shift operators, in descending order: the operators on the higher rows are acted on before those on lower ones. Those operators on the same row have equal precedence.

```
long   short   abs
**   shl   shr
*   /   div   rem
+   -
<   <=   >   >=   =   ¬=   is
¬
and
or
```

Example:

```
A and ¬B or A shr 7
```

is equivalent to

```
(A and (¬B)) or (A shr 7)
```

If no parentheses are used to enclose simple bits expressions within a bits expression containing operators of equal precedence, the expression is evaluated from left to right.

The evaluation of operators is completely specified in the description above. However, the evaluation of order of the left and right operands of a binary operator is unspecified and may occur in any order. because of this fact, a potential problem exists when a function implicitly changes a variable used elsewhere in the same expression. The result of the evaluation is unpredictable and such cases should be avoided.

## BITS ASSIGNMENT STATEMENTS

A bits assignment statement takes the following form:

```
<bits-variable> := <bits-expression>
```

where:

<bits-variable> is an identifier, a subscripted variable or a field designator of type bits and

<bits-expression> is as defined above.

Multiple assignment statements are legal and are of the same form as arithmetic multiple assignment statements. In other words, there must be a bits variable, not an expression on the left side of the assignment operator := .

Bits values may only be assigned to variables of type bits.

Examples of legal assignment statements:

```
A := #0000596B;
B := A or B shl 18;
B := #5943A260 shr 4;
C := B := (C and ¬B) shl 5;
```

PREDECLARED FUNCTIONS

Algol W provides two predeclared functions dealing with bits.

| Function Identifier | Function Type | Argument Type | Meaning |
|---|---|---|---|
| Bitstring | bits | integer | Binary representation of integer argument |
| Number | integer | bits | Integer corresponding to bits argument |

A discussion of the two's complement internal representation of integer values on System/370 type machines will be found in Appendix J. For example:

```
begin
   bits Y, Z;
   integer I, J;
   Y := #AB69;
   I := 4580;
   Z := Bitstring(I);
   J := Number(Y);
   I_W := 1;
   Write(Y, "is equivalent to the integer ", J);
   Write(I, "is equivalent to the bits value ", Z)
end.
```

The output from this program is:

```
#0000AB69  is equivalent to the integer 43881
4580  is equivalent to the bits value #000011E4
```

The assignment of the format variable I_W used in the above program is explained in the section "Basic Input and Output."

# PROCEDURES

Procedures are subprograms, which can either be part of an Algol W program or can stand alone as programs. If they stand alone as programs however, they cannot be executed directly. They are executed when called from other Algol W programs. Procedures which do not stand alone are executed when called from the program in which they have been declared.

One reason for using procedures is to avoid repeating code. If an algorithm is needed more than once in a given program, it is convenient to be able to use it without duplicating the identical code in several places. It is also much easier to debug a program if errors are confined to one segment of the program. In addition, a program's clarity and readability increase greatly if it is divided into subprograms each of which serves a specific purpose.

A procedure is viewed as a block. Therefore, any variables declared within a procedure are local to that block. In other words, they are accessible only within that procedure. Any variables used within a procedure must either be declared within that procedure, or be global (that is must be declared within one of the blocks which includes the procedure declaration). Since procedure declarations follow the same rules as variable declarations, any procedure A within a program can call another procedure B within the same program, provided that B is local or global (that is accessible) to the block which contains A. The global variables which a procedure can reference and the procedures it can call are determined by where the procedure is defined, not by where it is called. The same scope rules that apply to identifiers in general, apply to procedure identifiers in particular (see "Blocks" in the section "Statements").

There are two types of procedures in Algol W: proper procedures and function procedures.

## PROPER PROCEDURES

A proper procedure is a subordinate program which can be called by a procedure statement (see below). Two examples of predeclared proper procedures, which are already written and accessible to the Algol W user, are Read and Write.

## Declarations

The form of a proper procedure declaration is:

procedure <procedure-heading>; <statement>

where:

<procedure-heading> is either an identifier naming the procedure or an identifier followed by a parenthesized formal parameter list (to be discussed later);

the semicolon (;) serves to separate the <procedure-heading> from the <statement>; and

<statement> can be any legal <statement> as defined at the beginning of the section "Statements." In this context <statement> is often referred to as the <procedure-body>.

A procedure declaration within an Algol W program must be separated from the succeeding declaration or statement by a semicolon (;). Note that a procedure declaration, including the heading and body, constitutes a declaration within a program. Therefore, it must occur in the correct order: before the <statements> of the program or block. When a proper procedure is not being declared within a program but instead is standing alone to be called by other Algol W programs, a period (.) replaces the semicolon (;) after the <statement> part of the declaration.

## Proper Procedures without Formal Parameters

The following gives an example of a program that declares and then calls a proper procedure without a formal parameter list. A <procedure-statement> is a call to a proper procedure. The form of a <procedure-statement> in this case is simply:

<identifier>

where:

<identifier> is the name of the procedure given in the procedure heading.

```
begin
   real Sum, X, Y;  integer N;

   procedure Add;
   begin
      Sum := X + Y;
      Write(Sum)
   end Add;

   Read(N);
   for I := 1 until N do
   begin
      Read(X, Y);
      Add
   end
end.
```

If the following data are used:

```
3
1.2   4.5
5.6   10.8
7.    3.9
```

the output from the program is:

```
 5.700000
16.39999
10.90000
```

This program calls the procedure Add by a <procedure-statement> consisting only of the identifier naming the procedure, in this case:

```
Add
```

In the example above, no local variables are declared. All variables declared in the main program are global and thus accessible by all procedures within that program. In this case Sum, X and Y are global variables accessed by the procedure Add.

An important feature illustrated by this program is the special use of a comment without the reserved word 'comment'. The "Add" which appears after the 'end' indicating the end of the procedure declaration is a comment. It serves to clarify to the person reading the program where procedures and other blocks end. The rule is that if an identifier is inserted in between an 'end' and a reserved word, a semicolon (;) or a period (.) the identifier is regarded as a comment.

Proper Procedures with Formal Parameters

A procedure with parameters is more easily transferable among and within programs than one without any. Procedures with formal parameters do not have to rely on global variables. A main program or another procedure can call a procedure with formal parameters and provide values for variables at the time of the call.

The form of a proper procedure declaration with formal parameters is:

procedure <identifier> (<formal-parameter-list>); <statement>

where:

<identifier> and <statement> are as defined above;

the semicolon (;) separates the procedure heading from the procedure body; and

<formal-parameter-list> is a list of declarations of formal parameters, each separated by a semicolon (;), just like declarations in a block. The declarations can be of simple variables, arrays and procedures.

An identifier is local to a procedure if it is a formal parameter in the procedure declaration.

Warning: Of necessity, this example and others in this section introduce procedure parameters which are of the default "name" type. There are excellent reasons for using the non-default 'value', 'result' or 'value result' types. Users are recommended to read the Section "Parameter Passing Conventions" later in this chapter before attempting to code procedures.

Example:

```
procedure Add(integer  A, B);
begin
   real Sum;
   Sum := A + B;
   Write(Sum)
end;
```

A and B are formal parameters and, as such, valid identifiers within the procedure Add. However, they are not accessible outside the procedure. Sum is local to the procedure block in which it is declared. Note that the identifier Add is itself part of a declaration and is local to the block which contains it.

Each declaration in the formal parameter list has one of the following formats:

(1) Simple variable declarations

<simple-type> <identifiers>

where:

<simple-type> can be 'integer', 'real', 'long real', 'complex', 'long complex', 'bits', 'bits (32)', 'string', 'string (<integer-constant>)', 'logical', or 'reference (<record-class-identifier>)'; and

<identifiers> is a sequence of identifiers, each separated from the next by a comma.

Example:

```
procedure Add(real X, Y; integer N);
```

(2) Array declarations

<simple-type> array <dimension-spec> <identifiers>;

where:

<simple-type> and <identifiers> are as defined above; and

<dimension-spec> is a list of asterisks (*)'s, each separated from the next by a comma, each indicating a dimension of the array. Arrays declared as formal parameters in procedures do not specify the bounds of each dimension. The number of *'s given in the <dimension-spec> equals the number of dimensions in the array. By not requiring specification of array size in procedure declarations, Algol W provides the flexibility of allowing variable-length arrays to be passed as arguments to procedures.

Examples:

```
procedure Add(real array Score(*); integer N);
procedure Deter(real array Matrix(*,*); integer Col, Row);
```

(3) Procedure declarations

A procedure declaration in a formal parameter list takes one of the following forms:

procedure <identifiers>

<simple-type> procedure <identifiers>

where:

<identifiers> and <simple-type> are as defined above;

the first form indicates a proper procedure declaration; and

the second form indicates a function procedure declaration (see below). Note that a procedure declared as a formal parameter cannot have a formal parameter list or a procedure body.

The <procedure-statement> used to call a procedure with a formal parameter list is:

<identifier> (<argument-list>)

where:

<identifier> is the name of the procedure; and

<argument-list> is a sequence of items, each separated from the next by a comma, and corresponding in type and order to the parameters declared in the formal parameter list. The items can be variables, expressions, procedure identifiers, array identifiers and partial arrays. The <argument-list> must have the same number of entries as the <formal-parameter-list>. The <argument-list> is also known as the <actual-parameter-list>.

The following is an example of a program using a procedure with formal parameters and a procedure statement with a corresponding argument list. It is another version of the same procedure Add previously given without parameters.

```
begin
   real X, Y;
   integer N;

   procedure Add(real A, B);
   begin
      real Total;
      Total := A + B;
      Write(Total)
   end Add;

   comment Body of main program starts here;
   Read(N);
   for I:= 1 until N do
   begin
      Read(X, Y);
      Add(X, Y)
   end
end.
```

If the same data as above are used the output is identical. The difference is in the execution of the procedure statement. In both cases the body of the procedure Add is executed. However, in the program using the Add procedure with formal parameters, the names of the arguments are passed to the procedure and the argument items are evaluated in place of the formal parameters with which they correspond. In other words, the procedure with parameters is not dependent on specific global variables. The use of formal parameters allows greater flexibility and control of data flow than that provided by global variables.

An example of a procedure used as a parameter:

```
begin
   real array X(1::10);
    .
    .
   comment procedure Plot has a procedure
           as a formal parameter;
   procedure Plot(real procedure F; integer Size);
   begin
      for I := 1 until 10 do
      begin
         X(I) := X(I) + Size;
         X(I) := F(X(I))
      end
   end Plot;
   comment procedure Ellipsoid
           is a function procedure;
   real procedure Ellipsoid(real Z);
   begin
      real T;
       .
       .     ;
      T
   end Ellipsoid;
   comment Main program calls Plot, passing the
           procedure Ellipsoid as the argument
           corresponding to the formal parameter F;
   Plot(Ellipsoid, 1);
    .
    .
end.
```

Note that procedure F is not followed by a parameter list when it is declared as a formal parameter of the procedure Plot. However, when the main program passes Ellipsoid to procedure Plot, the number of arguments given in the function invocation F(X(I)) must match the number of formal parameters declared in the procedure Ellipsoid.

PARTIAL ARRAYS

The items in the <argument-list> can be variables (that is identifiers, subscripted variables, field designators or substring designators), procedures and partial or whole arrays.

Passing whole arrays as arguments in a procedure statement is legal if the number of dimensions in the argument array matches the number of *'s in the formal parameter array. It is also possible to pass part of an array.

For example, if a procedure is written with a one-dimensional array as a formal parameter, it is legal to pass one row or one column of a two-dimensional array as an argument to this procedure. The array argument is expressed as follows: the integer expression of the dimension desired is given and the remaining dimensions are specified by *'s.

Assume the following declarations in a partial program:

```
real array Matrix(1::10,5::10);
real array Cube(1::10,5::10,1::5);
procedure Find(real array A(*));
procedure Look(real array B(*,*));
```

Examples of legal procedure statements:

```
Find(matrix(6,*))
```

calls the procedure Find for the 6th row, all columns of array Matrix.

```
Look(Cube(*,4,*))
```

calls the procedure Look for the 4th column of the 3-dimensional array Cube.

```
Find(Cube(3,2,*))
```

calls the procedure Find for the 3rd row, second column of the 3-dimensional array Cube.

Note that in a procedure statement passing a partial array as an argument, the number of *'s in the argument list must equal exactly the number of asterisks in the formal parameter list. This means that the correct number of dimensions is being passed.

## FUNCTION PROCEDURES

A function procedure is a procedure that returns a value. The function itself is treated as an expression which means that its value can be assigned to variables with which it is assignment compatible and it can be used as an argument in procedure statements. Predeclared function procedures, such as Sqrt, Sin and Cos, are already written and accessible to the Algol W user. A function procedure is generally preferred over a proper procedure when only one value needs to be returned.

### Declarations

A function procedure declaration takes the following form:

<simple-type> procedure <procedure-heading>; <expression>

where:

<simple-type> can be 'integer', 'real', 'long real', 'complex', 'long complex', 'logical', 'string', 'string (<integer-constant>)', 'bits', 'bits (32)' or 'reference (<record-class-identifier>)';

<procedure-heading> is either an identifier naming the procedure or an identifier followed by a parenthesized formal parameter list;

the semicolon (;) is necessary to separate the <procedure-heading> from the <expression>, which in this context is called the <procedure-body>; and

<expression> can be any legal expression, including a <block-expression>, assignment compatible with the <simple-type>; <block-expression> is a block (see "Block Expressions" in the section "Constants, Variables, Expressions and Values") of the following form:

```
begin
   <declarations>;
   <statements>;
   <expression>
end
```

where:

<declarations> in the <block-expression> is optional; <expression> is as defined above.

Note that the <expression> in a <block-expression> may never be followed by a semicolon.

The value of the <expression> is the value of the function procedure.

## Function Procedures without Formal Parameters

There is no function procedure statement. A function invocation is an expression and is invoked using the following form:

```
<identifier>
```

where:

<identifier> is the name of the function procedure.

The following program gives an example of a function procedure without parameters and a Write statement which uses the function as an argument:

```
begin
   real X, Y;
   integer N;

   real procedure Max;
   if X < Y then Y else X;

   Read(N);
   for I := 1 until N do
   begin
      Read(X, Y);
      Write(Max)
   end
end.
```

This program reads a sequence of N pairs of numbers, and outputs the maximum of each pair. Using the data:

```
3
1.2   4.5
5.6   10.8
7     3.9
```

the output from the program is:

```
 4.500000
10.80000
 7.000000
```

Function Procedures with Formal Parameters

The same reasons and rules for using parameters in proper procedures apply to function procedures. The function procedure is invoked using the following form:

<identifier> (<argument-list>)

where:

<identifier> is the name of the function procedure and

<argument-list> is a sequence of items, each separated from the next by a comma, and corresponding in type and order to the parameters declared in the formal parameter list. The items can be variables, expressions, procedure identifiers, array identifiers and partial arrays. The <argument-list> must have the same number of entries as the <formal-parameter-list>.

The following program gives an example of a function procedure using parameters:

```
begin
   real Sum_Prod, Num1, Num2;
   integer N;

   real procedure Product(real A, B);
   begin
      real X;
      X := A * B;
      X
   end Product;

   Sum_Prod := 0;
   Read(N);
   for I := 1 until N do
   begin
      Read(Num1, Num2);
      Sum_Prod := Sum_Prod + Product(Num1,Num2)
   end;
   Write(Sum_Prod)
end.
```

This program reads a sequence of N pairs of numbers, obtains the product of each pair, adds the products together, and outputs the sum of the products.

## PARAMETER PASSING CONVENTIONS

When a procedure is called, the execution of the calling program is temporarily suspended and the body of the procedure is executed. The formal parameters take on values in various ways depending on the parameter passing conventions specified by the user. After execution of the procedure body has terminated, control returns to the calling program immediately following the procedure statement or function invocation.

There are several possible ways in which the arguments (that is actual parameters) can be passed to the procedure. Depending on the user's specifications, the formal parameters of a procedure (proper or function) can be call by name, call by value, call by result or call by value result.

### Call by Name

The programs above have all been written using name parameters. When a procedure with this type of parameters is called, the names (as opposed to the values) of the arguments are passed to the procedure. This has several important implications. First, the types of the arguments and the corresponding formal parameters must match exactly. It is not enough to have assignment compatibility. Note that for passing strings, this means that the string lengths must be equal. Second, when a procedure with a call by name formal parameter is invoked, the value of the argument variable or expression is not evaluated at the time of the procedure call. In other words, when the arguments are assigned to the formal parameters of the procedure (which occurs at the time of the procedure call), only the name is associated. The value of the variable or expression is only evaluated during execution of the procedure, each time the formal parameter is referenced. Third, changing the value of the formal parameter inside the procedure has the effect of changing the value of the corresponding argument. This process obviously involves added expense and is the reason that call by name should not be used in most Algol W programs. In fact, call by name should be used only if it is necessary to re-evaluate the value of the argument, that is actual parameter, in the course of execution of the procedure. Arrays and procedures can only be name parameters. Otherwise, call by value (see below) is a better choice. However, call by name is the default in Algol W: the user must specify explicitly if another parameter passing convention is desired.

An example of a partial program with a procedure using name parameters:

```
begin
    .
    .
   real procedure Integrate(long real X, Val);
   begin
      real Integral;
      Integral := 0;
      while X <= 1.0 do
      begin
         X := X + 0.1;
         Integral := Integral + Val*0.1
      end While_Loop;
      Integral
   end Integrate;
    .
    .
    .
   Write("Integral of Z**2 = ", Integrate(Z,Z**2))
end.
```

The above procedure illustrates the main advantage of using name parameters. The procedure Integrate can be evaluated many times with different values for Z and Z squared each time. Each time X or Val is encountered during the execution of the procedure Integrate, the value of Z or Z**2 is evaluated, respectively. If a parameter passing convention other than call by name were used, the value of X would be initialized to the value of Z at the time of the function invocation:

```
Integrate(Z,Z**2))
```

and would only change when incremented within the procedure. The value of Val would be initialized to the value of Z**2 at the time of the function invocation and would remain the same value throughout the execution of the procedure. Using name parameters allows Z**2 to be re-evaluated for new values of Z.

## Call by Value

Call by value is specified by inserting the reserved word 'value' after the <simple-type> in the formal parameter declaration. Call by value is a legal specification for formal parameters other than arrays or procedures. Arrays and procedures must be name parameters.

Call by value indicates that the value of the argument in the procedure statement or function call is to be passed to the procedure body. This means that the type of the argument need only be assignment compatible with the corresponding formal parameter. It does not have to match exactly. For example, it is legal to have a 'real value' formal parameter and a corresponding 'integer' argument. The length of a string argument can be less than or equal to the length of the

corresponding formal parameter. An argument passed to a call by value formal parameter can be an expression, a variable or a constant. When a procedure with a call by value parameter is invoked, the value of the argument is evaluated only once at the time of parameter assignment. Call by value is thus "safer" than any other calling type, because, if the formal parameter is changed in the procedure body, this has no effect on the value of the argument. Call by value is more efficient than call by name and should be used whenever only the value of the argument is needed and when it is not necessary to change the value. After the procedure has been executed, the value of the argument corresponding to the 'value' parameter still has the same value it did at the time the procedure was called initially. To retain the result of changes to the formal parameter without using name parameters, use call by result or value result (described later).

This program uses name parameters and the next similar program uses value parameters: they show the different effects of the two types.

```
begin

   procedure A(long real  S, T);
   begin
      Write(S, T);
      S := 5;
      Write(S, T)
   end;

   procedure B;
   begin
      long real Z;
      Z := 10;
      A(Z, Z**2);
      Write(Z)
   end;

   comment Main program starts here;
   B
end.
```

The output from the program above, which declares name parameters in procedure A, is:

```
10.00000    100.0000
 5.000000    25.00000
 5.000000
```

Using value parameters, the program would be coded:

```
begin

   procedure A(long real value S, T);
   begin
      Write(S, T);
      S := 5;
      Write(S, T)
   end;

   procedure B;
   begin
      long real Z;
      Z := 10;
      A(Z, Z**2);
      Write(Z)
   end;

   comment Main program starts here;
   B
end.
```

The output from the program above is:

```
10.00000    100.0000
 5.000000   100.0000
10.00000
```

Note that the change in the value of the formal parameter S did not change the value of the argument Z in the program where S is call by value, but did change the value of Z in the program specifying S as call by name. Notice also that Z**2 is evaluated only once when T is call by value and retains the value 100. When T is call by name, Z**2 is evaluated twice: once for each reference to T and has a new value the second time. Another interesting comparison: if real had been used in place of long real, the program with 'value' parameters would have produced the same output. However, the program with name parameters would have resulted in an error message:

```
Mismatched parameter
```

because Z**2 is always long real and thus, would not match its corresponding real formal parameter T.

## Call by Result

Call by result is specified by inserting the reserved word 'result' after the <simple-type> declaration of the formal parameter. The purpose of call by result is to assign the value of the 'result' formal

parameter at the end of the procedure to the corresponding argument in the procedure call. Therefore, the argument passed to a 'result' parameter must be a variable. It cannot be a constant or an expression. The reason for this rule is the same as the one requiring a variable on the left-hand side of the assignment operator := . Also, the type of the formal parameter must be assignment compatible with the type of the corresponding argument. The length of a string argument must be greater than or equal to the length of the corresponding formal parameter. Note that the direction of compatibility necessary is the reverse for 'value' parameters. This illustrates the principle that the length of the receiving string variable must be big enough to accommodate the other string. The value of an argument corresponding to a 'result' formal parameter is undefined until the formal parameter is assigned a value in the procedure body.

## Call by Value Result

When it is desirable both to initialize the value of a parameter and to save its modified value for use outside the procedure, it is possible to combine both call by value and call by result for a given parameter by specifying 'value result' after the <simple-type> in the formal parameter declaration.

The rules for value result arguments comprise both sets of rules for call by value and for call by result. The arguments must be variables, not expressions or constants. The assignment compatibility between the types of the formal parameter and the corresponding argument must exist in both directions. Note that this is still not as strict a requirement as the exact matching necessary for name parameters. For example a 'real' argument and a 'long real' formal parameter or vice versa would be legal for call by value result but not for call by name. However, string lengths of the formal parameter and corresponding argument must be equal.

The sequence of events in a call by value result specification is as follows: when the procedure is invoked, the value of the argument is evaluated, the value is passed to the corresponding formal parameter, the procedure body is executed and the final value of the formal parameter is then assigned to the argument. Call by value result is often similar in effect to call by name. However, when call by value result is used, the value of the argument is evaluated only once. In call by name, the value is evaluated each time the corresponding formal parameter is referenced in the procedure body.

Note that all four parameter passing conventions may be used freely within procedures, that is a given procedure (proper or function) may have different parameters using different conventions.

The following program illustrates some of the above parameter passing conventions in its procedures:

```
comment
   This program reads a given number Num of marks
   into an array Mark. It sorts the array Mark
   in descending order using the procedure
   Sort. It first prints the Marks in their original
   order, then in sorted order.  Next, it
   calculates and prints the median,
   using the function procedure Median. Finally, it
   calls the procedure Mode to determine whether or
   not there is a mode and, if so, prints its value. ;

begin
   integer Num;
   Read(Num);

   begin
      real array Mark(1::Num);
      logical Multiple;
      real Most_Freq_Mark;

      comment procedure Sort sorts an array of length
              N, in descending order. See sample
              program in the section "Arrays," for
              details of sorting method. ;

      procedure Sort(real array Score(*); integer value N);
      begin
         integer Large, First_Unor_Subs;
         for I := 2 until  N do
         begin
            Large := First_Unor_Subs := I - 1;
            for J := I until N do
            if Score(Large) < Score(J)
               then Large := J;
            if Large ¬= First_Unor_Subs then
            begin
               real Temp;
               Temp := Score(First_Unor_Subs);
               Score(First_Unor_Subs) := Score(Large);
               Score(Large) := Temp
            end
         end
      end Sort;

      comment
         Mode is a proper procedure which inputs two 'value'
         parameters Score and N, which don't change
         throughout the procedure. It returns two 'result'
         parameters Many and Hiscor, whose values are
         assigned to the arguments Multiple and
         Most_Freq_Mark, respectively.
         Mode searches the array and returns a logical
         value Many, indicating whether or not there is
```

```
      more than one mode, and a value for Hiscor, the
      mode, that is the unique value which occurs most
      frequently in the array. Curscore stands for the
      current element of the array. Oldscore is the
      previous element. Count is the number of times a
      Curscore has occurred in the already sorted array.
      Hifreq is the highest value of Count and shows
      the number of the times the current Hiscor has
      occurred.  The scheme is as follows: the 'for' loop
      goes down the ordered array, keeping track of how
      often the same Curscore occurs, by incrementing
      Count. When two scores differ, a check is made to
      see whether or not  the Curscore occurred more
      often than the current Hiscor. If so, the logical
      variable Many is set to 'false' and new values for
      Hiscor  and Hifreq are set. If there is a tie,
      Many is set to 'true' and the old values remain.
      If the Curscore occurred less often than Hiscor,
      nothing is changed. Now the procedure looks at
      the next score in the array, resetting the Count
      to 1. At the end of the procedure, Many indicates
      whether there is indeed one Hiscor and Hiscor is
      the mode. ;

   procedure Mode(real array Score(*); integer value N;
       logical result Many; real result Hiscor);
   begin
      real Curscore, Oldscore;
      integer Count, Hifreq;
      Oldscore := Score(1);
      Hifreq := 1;
      Many := true;
      Count := 1;
      for I := 2 until N do
      begin
         Curscore := Score(I);
         if Curscore ¬= Oldscore then
         begin
            if Count > Hifreq then
            begin
               Many := false;
               Hiscor := Oldscore;
               Hifreq := Count
            end else
            if Count = Hifreq then Many := true;
            Oldscore := Curscore;
            Count := 1
         end else
         Count := Count + 1
      end
   end Mode;

   comment
```

```
            Median is a function procedure which returns a
            real-valued expression. Its body consists of
            one conditional expression. If Number is even,
            the Median = the mean of the two middle scores.
            If Number is odd, the Median = the middle score. ;

         real procedure Median(real array Grades(*);
            integer value Number);
         if Number rem 2 = 0 then
            (Grades(Number div 2)
               + Grades(Number div 2 + 1)) / 2
         else
            Grades(Number div 2 + 1);

         comment Main program begins here ;

         comment The following three I/O commands to limit output
                 to 1 decimal place. See the section "Basic
                 Input and Output" ;
         R_Format := "A";  R_W := 5;  R_D := 1;
         for I := 1 until Num do
         begin
            Readon(Mark(I));
            if (I rem 5) = 1 then Write(" ");
            Writeon(Mark(I))
         end;
         Write(" ");
         Sort(Mark, Num);
         for I := 1 until Num do
         begin
            if (I rem 5) = 1 then Write(" ");
            Writeon(Mark(I))
         end;
         Write(" ");
         Write("Median= ", Median(Mark,Num));
         Mode(Mark, Num, Multiple, Most_Freq_Mark);
         if ¬Multiple then
            Write("Mode   =", Most_Freq_Mark)
         else
            Write("There is no unique mode")
      end
   end.
```

The following data:

```
   20
   67.1    86.2    61.9    81.7
   66.4    31.7    98.1    89.4
   19.8    54.7    70.5    99.2
   58.2    70.5    96.4    58.2
   72.1    77.0    22.4    58.2
```

produce the resulting output:

```
     67.1   86.2   61.9   81.7   66.4
     31.7   98.1   89.4   19.8   54.7
     70.5   99.2   58.2   70.5   96.4
     58.2   72.1   77.0   22.4   58.2

     99.2   98.1   96.4   89.4   86.2
     81.7   77.0   72.1   70.5   70.5
     67.1   66.4   61.9   58.2   58.2
     58.2   54.7   31.7   22.4   19.8

     Median = 68.8
     Mode   = 58.2
```

## RECURSIVE PROCEDURES

A recursive procedure is one which calls itself directly or indirectly. Recursive procedures are allowed in Algol W.

An example of a recursive function procedure is the following one which calculates the number N! (N factorial):

```
integer procedure Factorial(integer value N);
   if N = 0 then 1 else N * Factorial(N-1);
```

The above example is traditionally used to introduce recursion. However it should be pointed out that if an actual program required factorial evaluation many times during a run, it would be more efficient to initialise an integer array with the values of the factorial subscripts rather than re-evaluate the factorial each time.

The following program illustrates the use of a recursive procedure to solve the Towers of Hanoi problem. The Towers of Hanoi problem involves three rods with any number of disks. The disks are all of different sizes. To start with, all the disks are on one rod in order of size, the largest on the bottom. The problem is to move all the disks to another rod so that they are again in the same order. The constraints are that only one disk at a time can be moved, and that the disks on a rod always be in order of size, the largest on the bottom.

The recursive procedure Movedisks moves a given number of disks from Rod_A to Rod_C. The smallest numbered disk corresponds to the smallest (and thus highest) disk.

```
begin
   integer No_Disks;

   procedure Movedisks(integer value N;
      string(1) value Rod_A, Rod_C, Rod_B);
   begin
      if N > 0 then
      begin
         comment Moves top N-1 disks from Rod A to
                 temporary Rod B;
         Movedisks(N-1, Rod_A, Rod_B, Rod_C);
         comment Moves last (bottom) disk from
                 Rod A to Rod C;
         Write("Move disk number ", N, "from rod ",
            Rod_A, " to rod ", Rod_C);
         comment Moves same N-1 disks from Rod B
                 to Rod C;
         Movedisks(N-1, Rod_B, Rod_C, Rod_A)
      end
   end Movedisks;

   comment Main program starts here;
   Read(No_Disks);   I_W := 2;
   Write("Total number of disks is ", No_disks);
   Write(" ");
   Movedisks(No_Disks, "A", "C", "B")
end.
```

If the data file consists of:

```
4
```

meaning 4 disks, the output is:

```
Total number of disks is   4


Move disk number  1  from rod A to rod B
Move disk number  2  from rod A to rod C
Move disk number  1  from rod B to rod C
Move disk number  3  from rod A to rod B
Move disk number  1  from rod C to rod A
Move disk number  2  from rod C to rod B
Move disk number  1  from rod A to rod B
Move disk number  4  from rod A to rod C
Move disk number  1  from rod B to rod C
Move disk number  2  from rod B to rod A
Move disk number  1  from rod C to rod A
Move disk number  3  from rod B to rod C
Move disk number  1  from rod A to rod B
Move disk number  2  from rod A to rod C
Move disk number  1  from rod B to rod C
```

## EXTERNALLY DEFINED PROCEDURES

Procedures which are defined outside the Algol W program that calls them can either be Algol W procedures or else subprograms that follow O/S Type I (Fortran) linkage conventions. These are independently compiled and stand alone as programs (see the beginning of this chapter).

For further details of how to code and call such procedures and subprograms, see the section "External Linkages."

# STATEMENTS

A program in Algol W takes one of the following forms:

```
<statement>.
<proper-procedure-declaration>.
<function-procedure-declaration>.
```

where:

<statement> can be a <simple-statement>, an <iterative-statement>, or a <conditional-statement>; and

<simple-statement> can be any of the following:

```
<block>
<assignment-statement>
<procedure-statement>
<goto-statement>
<predeclared-procedure-statement>
<assert-statement>
<empty-statement>
```

Note that all <statements> (including blocks) are <simple-statements>, except for iterative and conditional statements.

For an explanation of <proper-procedure-declaration> and <function-procedure-declaration> see the section "Procedures." Note that a program which is just a procedure declaration cannot be executed directly. The corresponding procedures can be executed from other Algol W programs.

A discussion of each kind of <statement> follows.

## SIMPLE STATEMENTS

### Blocks

A block is composed of the following Algol W elements:

```
begin
   <declarations>
   <statements>
end
```

where:

<declarations> is a series of zero or more <declaration>s of any type; and

<statements> is a series of one or more <statement>s.

Each <declaration> and <statement> is separated from the following <declaration> or <statement> by a semicolon (;).

Since a block is a statement, a block with a period (.) following the 'end' constitutes an entire Algol W program. The ordering within a block is significant. Note that all <declarations> precede all <statements>. The <declarations> part of a block is optional. A block is a legal statement even if it contains no <declarations>. This type of block is called a trivial block. All blocks, however, must contain at least one statement. Notice that this definition of a block is a recursive one, that is a block which is itself a statement contains statements, which in turn can be blocks. Nontrivial blocks (that is blocks which contain declarations) can be nested up to eight levels deep. A compiler error message is returned if more than eight levels of nesting occur. The limit on nesting for trivial blocks is 29 levels. 'begin' and 'end' are both reserved words.

The syntax of Algol W requires every statement and declaration to be separated from its successor by a semicolon (;). The statement immediately preceding an 'end' need not be followed by a semicolon (;) since an 'end' alone does not constitute a statement. The Algol W rule is that the semicolon (;) is used to separate statements from each other, not to end a statement. For the same reason, a semicolon should not follow the 'begin'. If a semicolon is placed after the statement preceding an 'end', an <empty-statement> is interpreted as preceding the 'end' (see below for <empty-statement>). In general, this will not present any problems. However, if a semicolon appears after the last statement in a Case statement (see "Conditional Statements" later in this chapter) and the value of the <integer-expression> results in 1 greater than the total number of statements in the Case block, the <empty-statement> will be executed (meaning nothing will happen) and no error message returned, although an error may in fact have occurred. It is illegal to place a semicolon after an expression before an 'end'.

It is legal to begin a new block anywhere in an Algol W program. It is important to understand the concept of scope in Algol W. Scope refers to the range of blocks over which a given identifier is accessible. The scope of an identifier is determined by the following rules:

(1) An identifier is only meaningful within a block in which it is declared or defined. (Some identifiers do not have to be explicitly declared - see the section "Identifiers".)

(2) An identifier is local to a block if it is declared (or defined) within that block.

(3) An identifier is global to a block if the identifier is local or global to the block which contains this block and the same identifier is not declared in this block.

(4) An identifier local or global to a given block is not accessible to (that is has no significance within) a block which:

(a) is not contained by the given block or
(b) declares the same identifier within its block.

(5) If an identifier is declared more than once, when it is referenced the identifier declared within the most recently activated but not yet terminated block (with respect to the reference position) is accessed.

Example of a partial program:

```
begin
   real X;
   ..position 1
   begin
      real Y;
      ..position 2
   end;
   ..position 3
   begin
      real Z;
      ..position 4
      begin
         real X;
         ..position 5
         begin
            real P;
            ..position 6
         end
      end
      ..position 7
   end
   ..position 8
end.
```

Y is not accessible to positions 1, 3, 4, 5, 6, 7 or 8. Z is accessible to positions 4, 5, 6 and 7, but not to 1, 2, 3 or 8. The first X declared is accessible to all positions but 5 and 6, while the second X declared is only accessible to position 5 and 6. P is only accessible to position 6.

Assignment Statements

An assignment statement is the statement used to store a value into a storage location. It takes the form:

```
<variable> := <expression>
```

where:

<variable> is an identifier (see the section "Identifiers"), a subscripted variable indicating an element of an array (see "Subscripts" in the section "Arrays"), a field designator indicating a field of a record occurrence (see "Accessing Fields and Field Assignment Statements" in the section "Records and References") or a substring designator indicating a portion of a string (see "String Expressions" in the section "Strings");

<expression> is any legal expression whose type is compatible with the variable type.

Note that the assignment operator := is considered an operator, just like * (multiplication) and / (division). This implies that the order of evaluation of the <variable> and the <expression> is arbitrary. In general, this has no effect on the outcome of an assignment statement. Occasionally, however, it can be a problem. The following partial program using a function procedure with a value result parameter illustrates this point:

```
 .
 .
integer array A(1::10);

integer procedure Func(integer value result N);
begin
   N := N + 1;
   0
end Func;
 .
 .
 .
I := 1;
A(I) := Func(I);
Write(A(1), A(2));
 .
```

If the variable A(I) is evaluated first, then A(1) will be assigned the value 0. If the function Func(I) is evaluated first, then A(2) will be assigned the value 0. The user has no way of determining which evaluation will take place first.

The various legal assignment statements are given in the section "Arithmetic Expressions and Assignment Statements"; "Logical Assignment

Statements" in the section "Logicals"; "String Assignment Statements" in the section "Strings"; "Bits Assignment Statements" in the section "Bits"; and "Creating Records," "Accessing Fields and Field Assignment Statements,"and "Reference Assignment Statements" in the section "Records and References."

## Assignment Compatibility

Assignment compatibility for arithmetic expressions is explained in the section "Arithmetic Expressions and Assignment Statements." For all other than arithmetic types, compatibility means that both the type on the right side and the left side of the assignment operator := have to be the same. In other words, only string values can be assigned to string variables, reference values to reference variables, bits to bits variables, and logicals to logical variables. If the types are strings, the length of the variable on the left side of the assignment operator must be greater than or equal to the length of the string being assigned. If the types are references, the reference to be assigned must be 'null' or be pointing to a record occurrence belonging to one of the same record classes that the reference variable on the left side of the assignment operator is eligible to point to.

## Procedure Statements

See the section "Procedures."

## Goto Statements and Labels

Algol W provides the Goto statement to allow the programmer to change the flow of control in the program. Using Goto statements almost always makes a program less readable and more error prone. Therefore, the use of Goto statements is strongly discouraged. The Goto statement takes one of the following forms:

```
go to <label>
goto <label>
```

where:

'go to' and 'goto' are reserved words with the same meanings; and

<label> is any legal identifier elsewhere defined as a label.

The form of a <label> in front of a <statement> is:

<identifier> : <statement>

where:

<statement> can be any legal <statement>.

The Goto statement sends control to the statement labelled by the label specified in the Goto statement. It is illegal to place a label in front of a declaration, a 'begin', an 'end' or immediately following a 'then' or an 'else'. The reason for this restriction is to maintain a minimum of structured programming even with the use of a 'goto'. The restriction prevents jumping into the middle of an If, While or For statement. If a label appears before an 'end' and after a semicolon(;), the label is interpreted as labelling an <empty-statement> (see "Empty Statements" later in this chapter).

Note that a label is not declared. It is defined automatically by its usage. The <identifier> may be separated from the colon (:) by 0 or more blanks and the colon (:) may be separated from the <statement> by 0 or more blanks. It does not have to start in column 1 or any other specified column.

Example:

```
      .
      .
    Sum := 0;
    for I := 1 until N do
    begin
        .
        .
       for J := 1 until N do
       begin
           .
           .
          if abs(Denom(I,J)) > 1.'-5
          then
          Sum := Sum + Num(I,J)/Denom(I,J)
          else
          begin
             Write("J=", J, "I=", I);
             goto Exit
          end
       end For_J_Loop
    end For_I_Loop;
    Exit: Write("Current sum =", Sum)
end.
```

The conditions in the above program are sometimes considered justification for the use of a Goto statement in a structured program. This example has an error statement deeply nested in a loop, a cumulative Sum

which depends on all values of Num and Denom, and a normal ending of execution. If the Denom value is close to 0, the division should not take place. Instead, an error message giving the I and J values is printed, as well as the value of the current Sum. If an Assert statement were used instead of the If, execution would terminate in the event of a 0 value for the Denom element, without printing values of I, J, and Sum. The Goto statement is necessary to permit immediate exit from the loops, since Sum is only meaningful while the Denom values are not equal to 0. This illustrates the general principle that one can use a 'goto' to exit from deep within a loop. This program could be rewritten with While and If statements (see "Iterative Statements" and "Conditional Statements" later in this chapter) but perhaps not as well.

The scope of a label is determined in the same way as that of any variable identifier. A label identifier is local to a block if it is defined within that block. A label identifier is global to a block if the identifier is local or global to the block which contains the current block and the same identifier is not defined in the current block. An identifier local or global to a given block is not accessible to (that is has no significance within) a block which is not contained by the given block or defines the same identifier within its block.

```
begin
   integer I;
   I := 1;
   Loop: I := I + 1;
   if I > 6 then goto Stop;
   begin
      integer J;
      Newloop: J := I + 1;
      go to Loop;
      Stop:
   end;
   begin
      integer K;
      K := 2;
      I := K;
      goto Newloop
   end
end.
```

In the above example, the Goto statement 'goto Loop' is legal. However, 'goto Newloop' is not legal because Newloop is not defined within the block including that Goto statement. It is only defined in a parallel block. If that statement were 'goto Loop' instead, it would be legal since Loop is defined in the outermost block, meaning that all statements in the whole program have access to it. 'goto Stop' is not legal, because Stop is only defined in an inner block and the 'goto Stop' statement is in the main program. Note that the label Stop is actually labelling an <empty-statement> inserted after the semicolon (;) before the 'end'.

If there is more than one label represented by the same identifier, the flow of control passes to the statement within the most recently activated but not yet terminated block.

Example:

```
begin
   integer Num;
   Num := 1;
   Loop: Num := 2 * Num;
   if Num < 10 then goto Loop;
   begin
      integer M;
      Loop: M := 3 * Num;
      Num := Num + 1;
      if M > 18 then goto Stop
         else goto Loop;
      Stop:
   end
end.
```

The first 'goto Loop' statement passes control to the statement within the current block, that is the Loop statement 'Num := 2 * Num'. The second 'goto Loop' statement passes control to the Loop statement 'M := 3 * Num'.

Note that the use of labels and Goto statements (with some exceptions) could lead to programs that are not well structured. Therefore, the compiler responds with a warning message if any 'goto' is found in an Algol W program.

## Predeclared Procedure Statements

See the sections "Basic Input and Output," "Multiple Input and Output Streams," "Stream Directed Input and Output," and "Miscellaneous Topics" for details of the way predeclared procedures are provided. A list is given in Appendix E.

## Assert Statements

An Assert statement takes the following form:

```
assert <logical-expression>
```

where:

<logical-expression> is as defined in the section "Logicals"; and

'assert' is a reserved word.

The Assert statement evaluates the logical expression. If the value equals 'true', execution of the program continues normally. It is an error for an 'assert' logical expression to evaluate to 'false'. If this happens, the program terminates with an error message. See the "Assertion failed" error message under "Run-Time Error Messages" in Appendix C. Assert statements are useful in debugging.

A predeclared integer variable A_Count is initialized to 1 and incremented by 1 for each successful assertion. It may be inspected at any time by an executing program and in the event of a failed assertion its value is printed as part of the error message.

Example:

```
assert Length > 0
```

## Empty Statements

An empty statement takes the following form:

<empty>

where:

<empty> is the symbol for nothing.

An empty statement can be used anywhere in an Algol W program where a simple statement is appropriate. When a semicolon (;) precedes an 'end', it is interpreted as separating the previous statement from an empty statement. There are several possible uses of the <empty-statement>. One is to allow branching to the end of a loop. It is illegal to place a label in front of an 'end' since an 'end' is not a <statement>. Therefore, an example of a legal construction accomplishing the same purpose is:

```
for I := 1 until N do
begin
    .
   if <logical-expression> then goto Label;
    .
   ... ... ... ;
   Label:
end
```

Because a semicolon precedes the 'end', an <empty-statement> is inserted, and Label is labelling the <empty-statement>. The effect of

the Goto statement is to send control to the next cycle of the For statement loop. This use of the <empty-statement> parallels labelling the CONTINUE statement in a Fortran DO loop.

Another possible use of the <empty-statement> is in a situation where it is desirable to test a condition at the end of a loop. For example, the following construction (possible in some languages like Pascal, but not in Algol W):

```
repeat
   <statement>
until <logical-expression>
```

can be translated to Algol W as:

```
while begin <statement>; ¬<logical-expression> end do;
```

The <empty-statement> is inserted after the 'do' and before the ';' in this While statement loop. The block expression inside the 'while' loop performs the <statement> while the condition is not met, or in other words, until it is met. (See "While Statements" later in this chapter for an example of inserting a block expression in a 'while' loop.)

Another instance in which the <empty-statement> might be useful is in a Case statement where it is appropriate to do nothing in one of the cases.

Example:

```
case I of
begin
   D := -1;
   ;
   D := 1;
   D := 2
end
```

In the above example, if I equals 1, 3, or 4, D becomes -1, 1, or 2, respectively. If I equals 2, nothing happens. See "Blocks" earlier in this chapter for a problem caused by the unintentional use of an <empty-statement> in a Case statement.

## ITERATIVE STATEMENTS

Algol W has two types of iterative statements: While and For statements.

## While Statements

The While statement (also known as 'while' loop) takes the form:

```
while <logical-expression> do <statement>
```

where:

<logical-expression> has the value 'true' or 'false' and is as defined in the section "Logicals" and

<statement> can be any legal Algol W statement (including a block).

Both 'while' and 'do' are reserved words.

The purpose of the While statement is to execute the statement following the 'do' (possibly a block statement) repeatedly, as long as the given logical expression is 'true'. When the value of the expression is 'false', execution of the program continues with the statement succeeding the 'while' loop. Note that the While statement is exactly equivalent to the following:

```
Label: if <logical-expression> then
begin
   <statement>;
   go to Label
end
```

where it is understood that Label represents an identifier which is not defined at the place from which the While statement is entered.

Example of a 'while' loop:

```
begin
   real Num, Sum;
   Num := 10.0;
   Sum := 0;
   while Num <= 20.0 do
   begin
      Sum := Sum + Num;
      Num := Num + 0.2
   end;
   Write(Sum)
end.
```

The above program uses the 'while' loop to calculate the sum of all numbers from 10 to 20 at 0.2 intervals. It then prints the result Sum.

Because it is legal to insert a block expression (see "Expressions" in the section "Constants, Variables, Expressions and Values") wherever an expression may appear in an Algol W program, it is possible, and often very convenient, to have a 'while' loop execute one or more

statements before testing the logical expression. The following program illustrates this point:

```
begin
   integer Num, Sum;
   integer array A(1::100);
   Sum := 0;
   while
   begin
      Read(Num);
      Write(Num);
      Num ¬= 0
   end do
   begin
      Sum := Sum + Num;
      A(Num) := 0
   end;
   Write(Sum)
end.
```

The above program reads and writes a set of numbers until it reads in a zero. The 'while' loop adds the numbers and initializes the corresponding elements of the array A to zero. The 'while' loop is left when Num equals zero. Finally, the Sum is printed out. The program is equivalent to :

```
begin
   integer Num, Sum;
   integer array A(1::100);
   Sum := 0;
   Read(Num);
   Write(Num);
   while Num ¬= 0 do
   begin
      Sum := Sum + Num;
      A(Num):=0;
      Read(Num);
      Write(Num)
   end;
   Write(Sum)
end.
```

## For Statements

The For statement (also known as the 'for' loop) takes one of the following forms:

```
for <control-identifier> := <integer-expressions> do <statement>
```

for <control-identifier> := <integer-expression-1> step <integer-expression-3> until <integer-expression-2> do <statement>

for <control-identifier>:=<integer-expression-1> until <integer-expression-2> do <statement>

where:

<control-identifier> is any legal identifier;

<integer-expressions> are one or more <expressions> of type integer, each separated from the next by a comma;

<integer-expression-1> gives the initial value of the variable represented by the control identifier;

<integer-expression-2> gives the final value of the control variable;

<integer-expression-3> gives the increment value for the control variable; and

<statement> can be any legal Algol W <statement> (including a block).

'for', 'do', 'until' and 'step' are reserved words.

The purpose of the 'for' loop is to be able to execute a statement (possibly a block) repeatedly for different specified values of the control variable. In the first form of the 'for' loop, the <statement> following the 'do' is performed as often as there are expressions in the <integer-expressions>. The first time, the control variable has the first value listed, the second time, the second value, and the last time, the last value.

Example:

```
for I := 1, 4, 2*A, N do
begin
   Write(I);
   B(I) := 0
end
```

The above statement writes the specified values and sets the corresponding array elements to 0. Note that both A and N must have been declared as integers to be legal expressions in the For statement.

The second form of the 'for' loop starts performing the <statement> following the 'do', with the value of the control variable set at the value of <integer-expression-1>, and continues to execute the <statement>, incrementing the value of the control variable each time by the value of <integer-expression-3>, until the control variable reaches the value given by <integer-expression-2>. The last time the <statement> is

performed occurs when the control variable is set either at the value of <integer-expression-2> or at the integer value less than <integer-expression-2> by an amount less than <integer-expression-3>. Note that the value of <integer-expression-3> following 'step' should not be equal to 0, or the statement will never terminate. It can, however, be negative. In other words, this is a legal For statement:

```
for I := 10 step -2 until 0 do Write(I)
```

which produces the following output:

```
10
8
6
4
2
0
```

An example of a For statement which stops executing the <statement> part, with the value of the control variable less than the value of <integer-expression-2>:

```
for I := 1 step 2 until 10 do
   Write(I)
```

produces the following output:

```
1
3
5
7
9
```

The third form of the 'for' loop is really a subset of the second form. It just assumes that the 'step' value equals 1.

Example:

```
for I := 1 until N do
begin
   Read(Score(I));
   if Score(I) > Big then Big := Score(I)
   else
   if Score(I) < Small
      then Small := Score(I)
end
```

This 'for' loop reads N values and makes N comparisons. I takes on the values 1, 2, ..., N, that is all values between 1 and N inclusive, at 1 step intervals.

The <control-identifier> is not declared. It is defined by its usage in the For statement. A control identifier is local to the statement following the 'do'. It cannot be accessed outside of this statement.

```
for I := 1 until N do write(A(I))
```

The control identifier I is meaningful only within the For statement. If I is used in another part of the program, it does not represent the same variable, and has to be defined in one of the ways discussed in the section "Identifiers." The value of the control variable cannot be changed within the <statement> following the 'do'. Similarly the limit <integer-expression-2> and the step size <integer-expression-3> are evaluated once only, on entry to the 'for' loop.

If the <integer-expression-1> equals the <integer-expression-2>, then the 'for' loop is executed exactly once. For example:

```
for I := 2 until 2 do <statement>
```

is exactly equivalent to

```
for I := 2 do <statement>
```

If <integer-expression-2> is less than <integer-expression-1> and the 'step' value is positive, the <statement> following 'do' is not executed at all and control passes to the statement following the 'for' loop. The same result occurs for the reverse conditions, that is <integer-expression-2> is greater than <integer-expression-1>, and the 'step' value is negative.

Example of a partial program:

```
A := 4;
B := 2;
C := 3;
for I := B + C until A do
<statement>;
Write(D(A), D(B), D(C))
```

The above <statement> does not get executed at all; control passes immediately to the Write statement. Note that A, B, and C must have been declared as integers.

Example of 'for' loop:

```
begin
   integer Sum;
   Sum := 0;
   for I := 1 until 100 do
      Sum := Sum + I
end.
```

The above program adds all the integers from 1 to 100 inclusive.

## CONDITIONAL STATEMENTS

Algol W has two types of conditional statements: If statements and Case statements.

### If Statements

The If statement takes one of the following forms:

```
if <logical-expression> then <statement>
if <logical-expression> then <simple-statement> else <statement>
```

where:

<logical-expression> is as defined in the section "Logicals";

<simple-statement> can be any legal <statement> (including a block), except for an iterative or a conditional statement and

<statement> is as defined at the beginning of the current chapter.

'if', 'then' and 'else' are reserved words.

The purpose of the first form of the If statement is to execute a statement (possibly a block) only if a given condition is true. If the <logical-expression> is 'true', the statement following the 'then' is executed. If the <logical-expression> is 'false', the statement following the 'then' is skipped and control passes to the statement following the If statement. Note that the <statement> following the 'then' in the first form of the If statement can be any legal statement, not just a simple one. In other words, an iterative statement or a conditional statement can follow the 'then'.

Examples of legal If statements:

```
if (A > B) and (Next(Place) ¬= null) then
   if Name(Place) = Coursename then
      Write(Coursename)

if Line(K|1) ¬= " " and K <= 79 then
for I := 0 until K do
begin
   Write(Line(I|1));
   Letters := Letters + 1
end
```

```
if (A > B) and (Next(Place) ¬= null)
   then Write(Name(Place))
```

The purpose of the second form of the If statement is to execute a statement (possibly a block) if the logical expression is true and to execute a different statement if the logical expression is false. Note that the statement following the 'then' in this form of the If statement must be a simple one. However, any statement may follow the 'else'.

Examples of legal If statements:

```
if Grade > Big
   then Big := Grade
else if Grade < Small
   then Small := Grade
```

In the above, a simple assignment statement follows the 'then' and a conditional statement follows the 'else'.

```
if Switch_Them then
begin
   Temp := Next;
   Next := First;
   First := Temp
end
else
   Write("OK as is")
```

Here a block, which is a type of simple statement, follows the 'then'.

Example of an illegal If statement:

```
if Number(Place) = Num
then
if Oldplace = Place
then First := Next(Place)
else
   Next(Oldplace) := Next(Place)
else
   Write("No record with number", Num)
```

The statement above is illegal because a conditional statement, which is not a simple statement, follows the 'then'. The If statement can be modified to be correct syntactically and still express the same meaning:

```
if Number(Place) = Num then
begin
   if Oldplace = Place
   then First := Next(Place)
      else Next(Oldplace) := Next(Place)
end
else Write("No record with number", Num)
```

With the insertion of 'begin' and 'end', the inner conditional statement is enclosed in a block, which is a simple statement.

Since 'then' and 'else' clauses are part of an If statement, it is not legal to have a semicolon preceding a 'then' or an 'else' (except if a 'comment' occurs immediately in front of a 'then' or an 'else').

It is not legal to have a label immediately following a 'then' or an 'else'. This rule forces a minimum of structured programming by not permitting a jump into the middle of an If statement. It is legal, however, to have a label within a block, following a 'then' or an 'else'. This causes the label to be local to the given block. Thus, Goto statements within that block can jump to the labelled statement but Goto statements outside the block cannot.

A sample program using If statements:

```
begin
   real A, B, C, X1, X2;
   while
   begin
      Read(A, B, C);
      Write(A, B, C);
      A < 1000
   end do
   begin
      if A = 0 then
      begin
         if B ¬= 0 then
         begin
            X1 := -C/B;
            Write("X =", X1)
         end
         else Write("Meaningless")
      end
      else
      begin
         if B**2 - 4*A*C < 0
         then Write("Imaginary roots")
         else
         begin
            X1 := (-B + Sqrt(B**2 - 4*A*C)) / (2*A);
            X2 := (-B - Sqrt(B**2 - 4*A*C)) / (2*A);
            Write("X =", X1, "or", X2)
         end
      end
   end
end.
```

The program above solves quadratic equations for given values of A, B and C. It checks for 0 values which would make the equation linear or meaningless. It only computes equations with real roots. It ends when

A has a value greater than or equal to 1000. Using the following sample data:

```
5.     4.     2.
3.     8.     1.
0.     4.     3.
0.     0.     5.
1001.  0.     0.
```

the output from the program is:

```
      5.000000        4.000000        2.000000
Imaginary roots
      3.000000        8.000000        1.000000
X=    -0.1314829  or     -2.535183
            0         4.000000        3.000000
X=    -0.7500000
            0               0         5.000000
Meaningless
      1000.000              0               0
```

## Case Statements

The Case statement takes the form:

case <integer-expression> of begin <statements> end

where:

<integer-expression> can be any legal expression of type integer; and

<statements> are a list of any type of <statement>s, each separated from the next by a semicolon (;).

'case', 'of', 'begin' and 'end' are all reserved words.

The purpose of the Case statement is to offer more than the two choices available in the If statement. The value of the integer expression indicates which of the <statements> listed is to be executed. If the value equals 1, then the first statement is executed, if the value equals 2, then the second statement is executed, etc. The value must be greater than zero and must not be greater than the number of <statements> listed.

A sample program using a Case statement:

```
begin
   real X;  integer Num;
   X := 0;
   while
   begin
      Read(Num);
      Num ¬= 0
   end do
   case Num of
   begin
      X := X + 1;
      X := X - 1;
      X := X**2
   end;
   Write(X)
end.
```

The sample program adds 1 to X if Num equals 1, subtracts 1 from X if Num equals 2 and squares X if Num equals 3. When Num equals 0, the 'while' loop is left and the final value of X is printed.

Note that each statement within a Case statement can itself be a block or even a nonsimple statement, that is a conditional or iterative statement.

An example of Case statement:

```
case (I + J) of
begin
   begin   comment (I + J) = 1;
      X := X + 1;
      Y := Y + 1
   end;
   begin   comment (I + J) = 2;
      X := X - 1;
      Y := Y - 1
   end;
   begin   comment (I + J) = 3;
      X := 0;
      Y := 0
   end;
   begin   comment (I + J) = 4;
      X := X**2;
      Y := Y**2
   end
end
```

See "Blocks" earlier in this chapter, for a potential problem caused by an <empty-statement> being inserted in a Case statement.

# RECORDS AND REFERENCES

A record is a group of simple variables placed together. A record value is an ordered set of values each of which may be a different simple type. The word "record" is ambiguous in that it is often used to denote two distinct concepts: record class and record occurrence. A record class is defined as a group of variable types placed together. Each variable type specified is known as a field. The value of each field is of the type declared for that field. A record class defines a class of record occurrences all with the same fields but with different values for each field. Note that the fields cannot be of structured variable types, that is there are no record classes of arrays or record classes of records.

## RECORD CLASS DECLARATIONS

A record class declaration takes the following form:

record <identifier> (<simple variable declarations>)

where:

<identifier> is any legal name denoting the record class; and

<simple-variable-declarations> are one or more legal declarations of variables of type integer, real, long real, complex, long complex, logical, string, bits or reference.

Example:

```
record Personnel(string(20) Name; integer Socsecno, Age;
   reference(Personnel) Next);
```

declares a record class Personnel with four fields: Name, Socsecno, Age and Next.

No more than fifteen record classes can be declared within a given Algol W program.

REFERENCES

Since there is usually more than one record occurrence per record class, it is not enough to specify the field name of a record class in order to access a given record occurrence. Therefore, to access a particular record occurrence, one must use a variable of type reference. A reference variable points to occurrences of the record class defined in the declaration of the reference. A field of a record occurrence is accessed by giving the field name and the reference to the record class desired.

A reference declaration takes the following form:

reference (<record-class-identifiers>) <identifiers>

where:

<record-class-identifiers> is a list of all the record classes (previously declared) which the reference can point to, each one separated from the next by a comma, and

<identifiers> is a list of identifiers specifying the reference variables being declared, each one separated from the next by a comma.

Example:

```
reference(Personnel) List;
```

CREATING RECORDS

The declarations of record classes alone do not set aside storage space as is the case with other variable declarations. They just define the structure of the record class. In order to assign storage space for record occurrences, the following type of assignment statement is used:

<reference-variable> := <reference-expression>

where:

<reference-expression> is either of two forms:

<record-class-identifier>
<record-class-identifier> (<expressions>)

<reference-variable> is either an identifier, a subscripted variable, or a field designator of type reference (see "Accessing Fields and Field Assignment Statements" later in this section);

<record-class-identifier> is as defined above; and

<expressions> is a list of expressions, one for each field specified in the record class declaration and each matching the type of its corresponding field. Each expression is separated from the next by a comma. The <empty> symbol can be used as one or more of the expressions listed. In other words, if values of only some fields are to be assigned, the other fields can be skipped over by putting nothing in between the commas separating the values of the fields.

The <reference-variable> must have been previously declared a reference to the record class given.

In either form of the assignment statement, the value of the reference variable is the address of the record occurrence defined. If the first form is used, an empty record occurrence is created. If the second form is used, a record occurrence with the specified values is created.

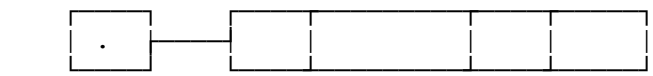
Using the these declarations in a partial program:

```
record Personnel(string(20) Name; integer Socsecno, Age;
   reference(Personnel) Next);
reference(Personnel) List;
```

the statement:

```
List := Personnel;
```

creates the following empty record occurrence:




and the statement:

```
List := Personnel("Joe Smith", 562487809, 35, null);
```

would create the first record occurrence in the following figure, except that the Next field would contain the value 'null' (a reserved word) instead of a pointer.

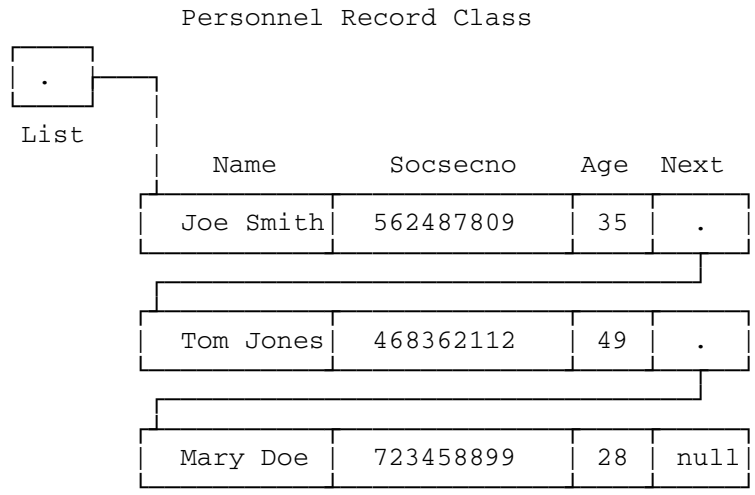
In order to create a second record occurrence in this list, one would say:

```
Next(List) := Personnel("Tom Jones", 468362112, 49, null);
```

which has the effect of creating a new record occurrence whose address is stored in the Next field of the first record occurrence.

```
    Next(Next(List)) := ("Mary Doe", 723458899, 28, null);
```

creates the final record occurrence in the list.




Now the entire list is created as shown. The Next field of each record occurrence in this list points to the next one in the list. The last has a 'null' value in its Next field indicating that the end of the list has been reached.

Example of the use of the <empty> symbol in record creation:

```
    List := Personnel("Bill Evans",,,null);
```

creates a record occurrence in the record class Personnel with no values for the Age and Socsecno fields. These fields remain undefined.

Example of a <reference-expression> as one field value in <expressions>:

```
    Next(List) := Personnel("Tom Jones", 468362112, 49,
    Personnel("Mary Doe", 723458899, 28, null) );
```

could have been used to create the last two record occurrences in the list above (after only the Joe Smith record occurrence existed) with one statement.

## ACCESSING FIELDS AND FIELD ASSIGNMENT STATEMENTS

In order to access a field of a record occurrence, the following form, called a <field-designator>, is used:

<field name> (<reference-variable>)

where:

<field-name> is an identifier declared in the record class declaration, referring to a given field; and

<reference-variable> is an identifier, a subscripted variable or a field designator of type reference.

For example, given the records as shown above,

```
Name(Next(List)) = "Tom Jones"
```

has the value 'true'.

A field of a record occurrence is assigned a value in the same way as any variable of that field type. The form is:

<field-designator> := <expression>

where:

<field-designator> is as defined above; and

<expression> can be any legal expression which is assignment compatible with the field on the left-hand side of the operator := . Multiple assignment statements are legal as long as the <expression> part is never on the left-hand side of the operator := .

Examples of field assignment statements:

```
Age(List) := 36
```

updates the age of Joe Smith's record.

```
Socsecno(Next(Next(List))) := 723458999
```

changes Mary Doe's social security number. Note that Next(Next(List)) is a single reference variable.

REFERENCE ASSIGNMENT STATEMENTS

A reference assignment statement takes one of the following forms:

```
<reference-variable> := <reference-variable>
<reference-variable> := null
```

where:

<reference-variable> is as defined above and

'null' is the reference constant indicating that the <reference-variable> is currently not pointing to any record occurrences.

The result of the first type of reference assignment statement is to cause both references to point to the same record occurrence.

Multiple assignment statements are legal as long as 'null' is never on the left-hand side of the operator :=.

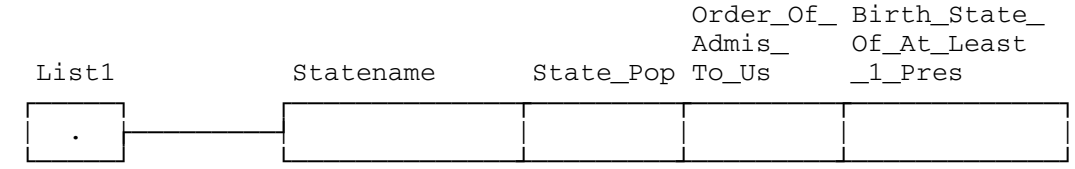
Assume the following declarations and assignments in a partial program:

```
record State(string(20) Statename;
   integer State_Pop, Order_Of_Admis_To_Union;
   logical Birth_State_Of_At_Least_1_Pres);

reference(State) Statelist, List1, List2;
```

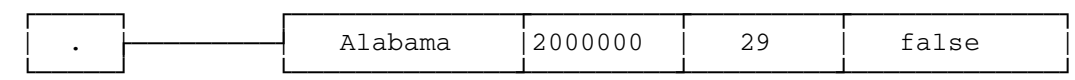
Sets aside three reference storage locations.

```
List1 := State;
```

Creates an empty record occurrence of record class State whose address is stored in List1.

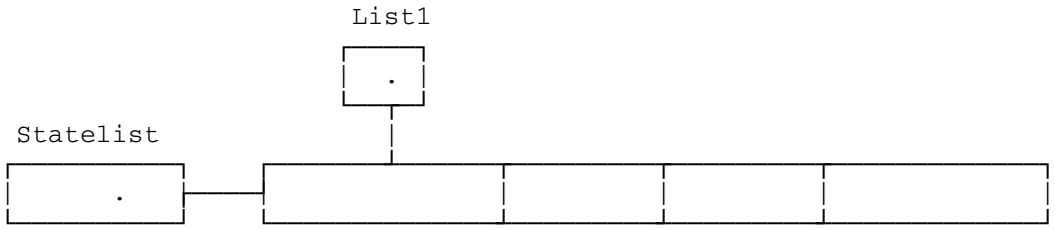



List2 := State("Alabama",2000000,29,false);
creates the following record occurrence whose address is stored in List2.

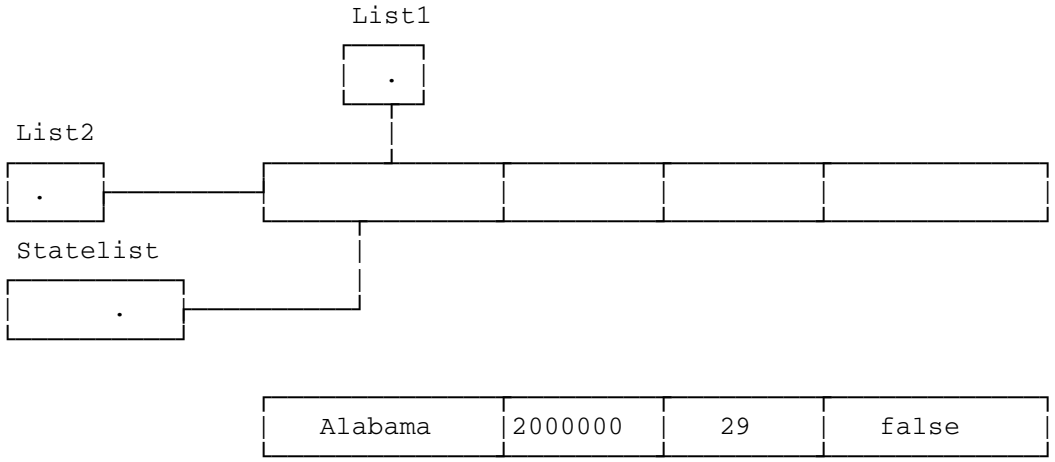


Examples of reference assignment statements:

Statelist := List1;
stores the reference value of List1 in the variable Statelist, causing both to point to the empty record occurrence.




List2 :=List1;
stores the reference value of List1 in the variable List2, causing all three reference variables to point to the empty record occurrence, and no reference pointing to the record occurrence with values in it.




Since there are no pointers to the Alabama record occurrence, it is now inaccessible.

## REFERENCE ARRAYS

The following program makes use of a reference array variable to read and store 50 sets of values for record occurrences belonging to the record class State declared above. The input data for the program has the States ordered according to Order_Of_Admis_To_Union, that is order of admission to the United States.

```
begin
   record State(string(20) Statename; integer State_Pop,
      Order_Of_Admis_To_Union;
      logical Birth_State_Of_At_Least_1_Pres);
   reference(State) array Statelist(1::50);
   string(20) Name;
   integer Pop, Order;
   logical Home_State;
   for I := 1 until 50 do
   begin
      Read(Name, Pop, Order, Home_State);
      Statelist(Order) := State(Name, Pop, Order, Home_State)
   end
end.
```

The 50 record occurrences are not stored contiguously but the references to the record occurrences are in order from 1 to 50 according to order of admission.

## LINKED LISTS

Using arrays of references is one way to store record occurrences. The other way is by making use of a data structure called linked lists. A linked list is a list of record occurrences belonging to a record class, which includes a reference variable as one of its fields. The examples above, in which Next was a field of the record class declared, were examples of linked lists. Each record occurrence in the list pointed to the succeeding record occurrence.

### Insertions of Records at the Beginning of a List

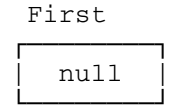
The following program makes use of the procedure Inserttop to insert records at the beginning of a list.

```
begin
   record Course(integer Num; string(20) Name;
      reference(Course) Next);
   reference(Course) First;

   procedure Inserttop(integer value Count;
      string(20) value Coursename;
      reference(Course) value result Follow);
   Follow := Course(Count, Coursename, Follow);

   comment Main program starts here;
   First := null;
   for I := 1 until 4 do
```

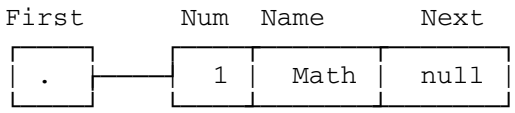
```
        Inserttop(I, "Math", First)
    end.
```

This example program goes through the following steps:

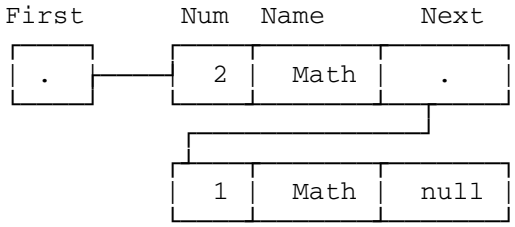
Initialize first to null:



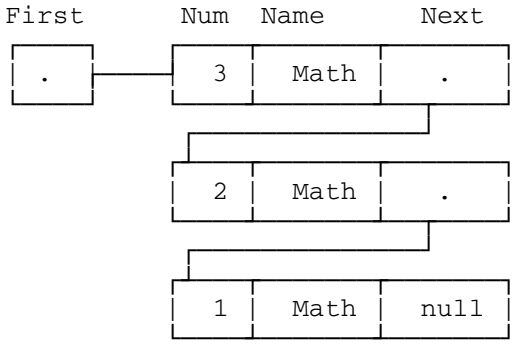

Create first record:



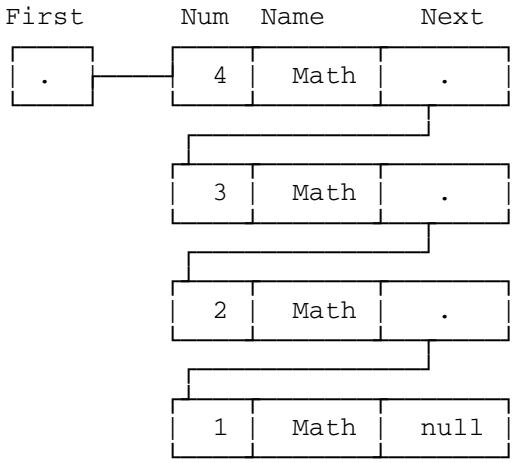

Add a second record to from a linked list chain:




Add a third record to the list:




Finally add a fourth record:

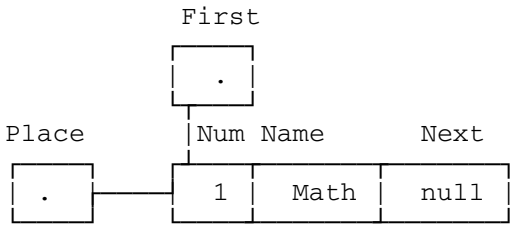



Insertions of Records at the End of a List

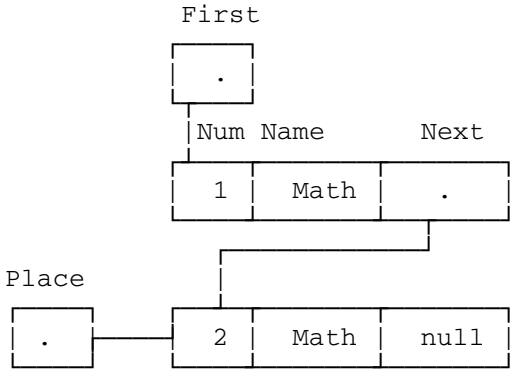
The following program makes use of the procedure Insertend to insert records at the end of a list.

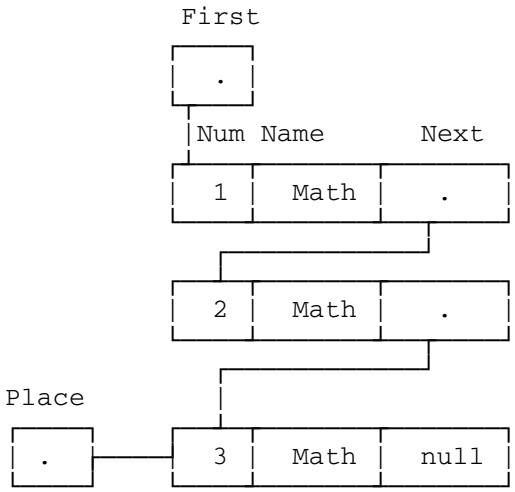
```
begin
   record Course(integer Num; string(20) Name;
      reference(Course) Next);
   reference(Course) First, Place;

   procedure Insertend(integer value Count;
      string(20) value Coursename;
      reference(Course) value result Last);
   Last := Next(Last) := Course(Count, Coursename, null);

   comment Main program begins here;
   Place := First := Course(1, "Math", null);
   for I:= 2 until 4 do
      Insertend(I, "Math", Place)
end.
```

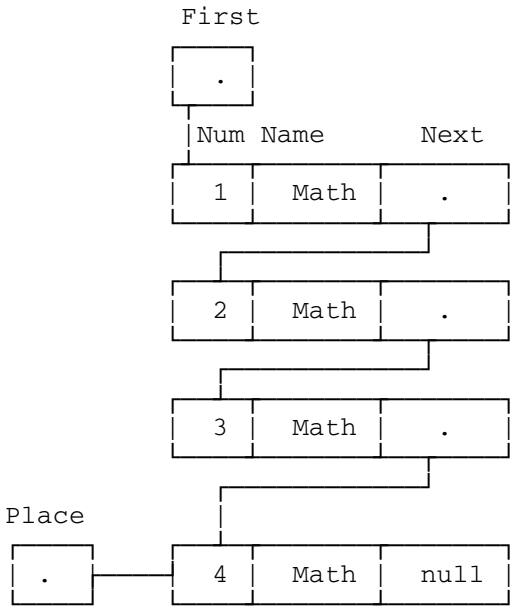
This sample program goes through the following steps:



Start 'for' loop






Inserting Records in Sequential Order into an Ordered List and Deleting Records from a List

The following program makes use of two procedures: procedure Inseq inserts records in sequential order into an ordered list and procedure Delete deletes records from the list.

```
 begin
    record Course(integer Number; string(20) Name;
       reference(Course) Next);
    reference(Course) First;
    string(20) Coursename;
    integer Num, N;

    comment procedure Inseq inserts records sequentially
            into an ordered list;

    procedure Inseq(integer value Key;
       string(20) value Title;
       reference(Course) value result Start);
    begin
       comment Oldplace is kept as a trailing pointer
               to facilitate insertion. It always points
               to the record before the one pointed to by
               Place;
       reference(Course) Oldplace, Place;

       comment initialize;
```

```
      Oldplace := null;
      Place := Start;
      comment While statement traverses list from low to high,
              stopping as soon as new entry belongs before
              Place, or when end of list is reached;
      while (Place ¬= null) and (Key > Number(Place)) do
      begin
         Oldplace := Place;
         Place := Next(Place)
      end Find_Place;

      comment Four cases:
         1 - Place, Oldplace both not null -> Insert
               between them
         2 - Place = null, Oldplace ¬= null -> Insert
               at end of list
               (Oldplace points to last entry)
         3 - Place ¬= null, Oldplace = null -> Insert
               at beginning of list (Place = first)
         4 - Place = Oldplace = null -> Insert in empty list
      Note that 2 is a special case of 1 and 4 is a special
      case of 3;

      if Oldplace = null
      then Start:= Course(Key, Title, Start)
      else Next(Oldplace) := Course(Key, Title, Place)
   end Inseq;

   comment procedure Delete deletes records from a list;
   procedure Delete(integer value Key;
      reference(Course) value result Start);
   begin
      comment Oldplace is kept as a trailing pointer, to
              facilitate deletion. It always points to the
              record before the one pointed to by Place;
      reference(Course) Oldplace, Place;
      comment Initialize Oldplace, Place to Start
              record in list;
      Oldplace := Place := Start;
      comment If list empty, write error message;
      if Place = null
      then Write("Nothing left in list")
      else
      begin
         comment While statement loop searches for given record,
                 or end of list -- moves Place and
                 Oldplace down list;
         while (Next(Place) ¬= null)
            and (Number(Place) ¬= Key) do
         begin
            Oldplace:= Place;
            Place := Next(Place)
         end Find_Place;
```

```
            comment If find record to be deleted, then put
                    address of following record in Next field
                    of previous record. Now only Place points
                    to the deleted record;
            if Number(Place) = Key then
            begin
               comment Check if deletion at
                       beginning of list;
               if Oldplace = Place
               then Start := Next(Place)
               else Next(Oldplace) := Next(Place)
            end Pointer_Change
            else Write("No record with number", Key)
         end Else_Block
      end Delete;

      comment Main program starts here;
      comment Initialize;
      First := null;
      comment Read first card to see how many record
              occurrences to be inserted in sequential order;
      Read(N);
      comment For N set of values, read and create
              corresponding records by calling procedure Inseq;
      for I := 1 until N do
      begin
         Read(Coursename, Num);
         Inseq(Num, Coursename, First)
      end;

      comment Read next card to see how many record
              occurrences to be deleted;
      Read(N);
      comment For N sets of values, read Num and delete
              corresponding records by calling procedure
              Delete;
      for I := 1 until N do
      begin
         Read(Num);
         Delete(Num, First)
      end
   end.
```

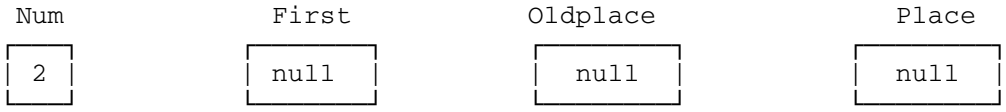
Using the following data as input:

```
4
"Math" 2
"Math" 5
"Math" 1
"Math" 4
5
2
5
1
4
2
```

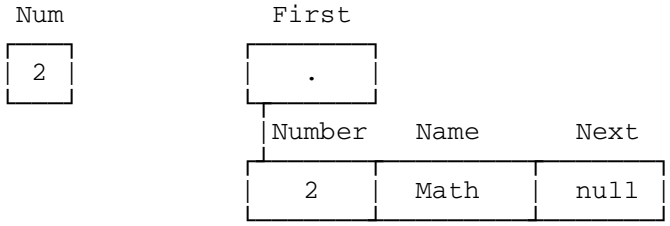
the program would go through these steps:

Insertion part

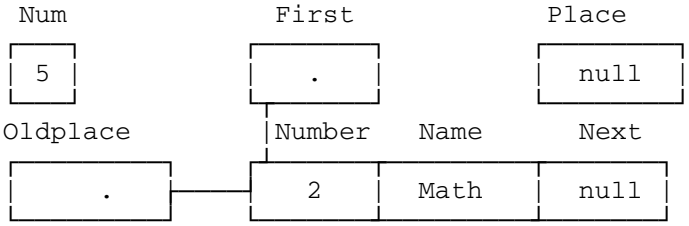
At end of 'while' loop in procedure Inseq during 1st pass through 'for' loop of main program:



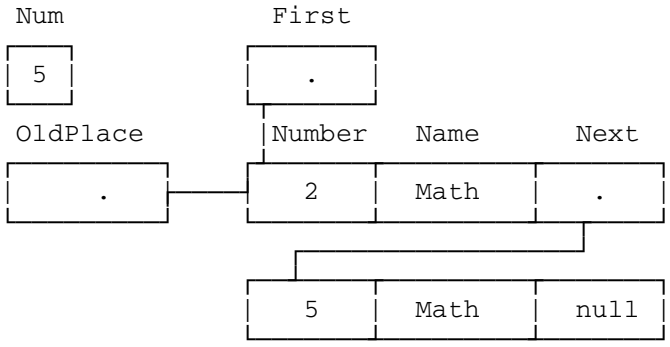

At end of 1st pass through 'for' loop, i.e., at end of first call to procedure Inseq:



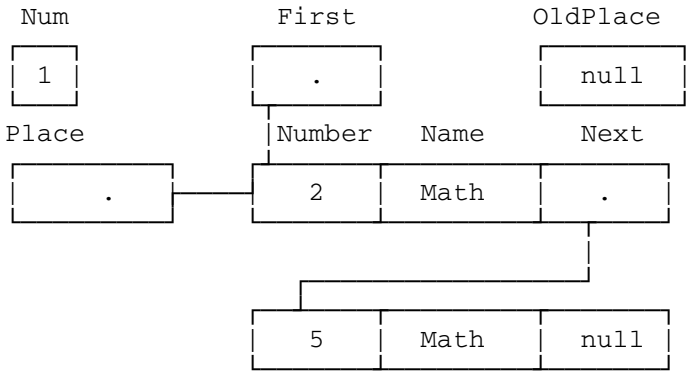

At end of 'while' loop in procedure Inseq during 2nd pass through 'for' loop of main program:



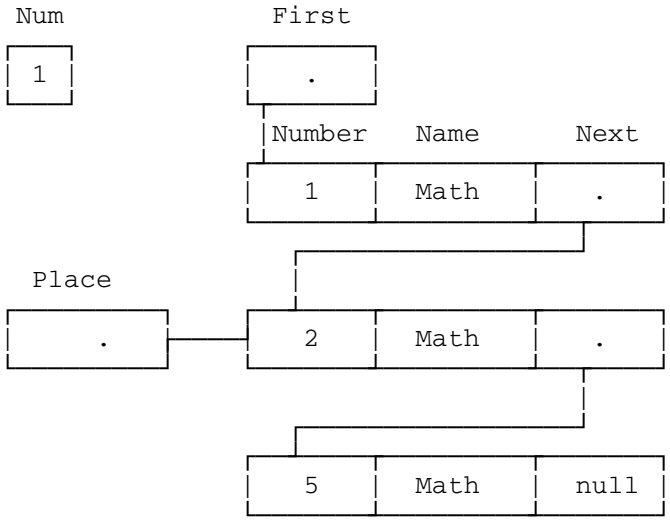
At end of 2nd pass through 'for' loop, i.e., at end of 2nd call to procedure Inseq:



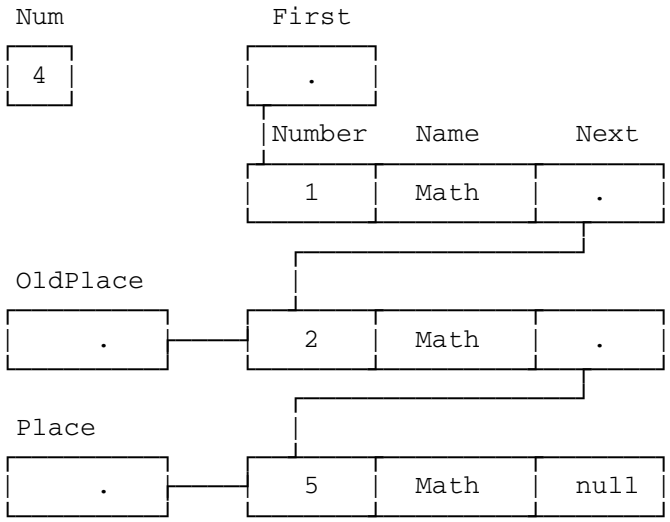

At end of 'while' loop in procedure Inseq during 3rd pass through 'for' loop in main program:



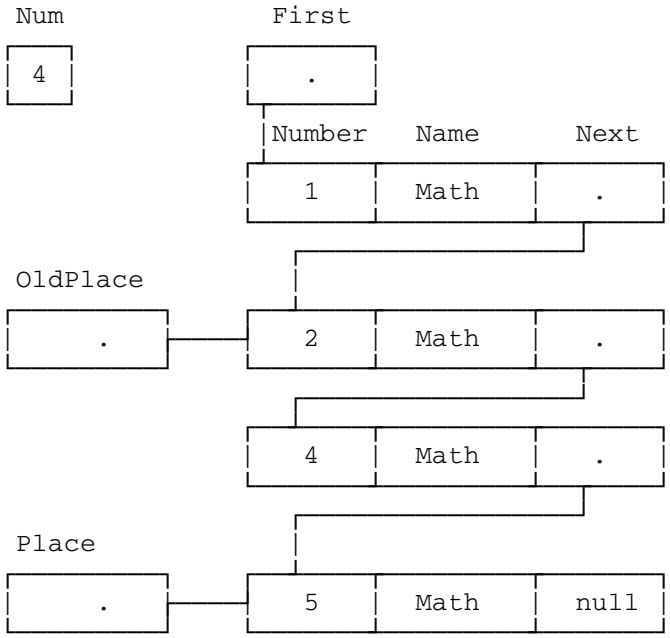
At end of 3rd pass through 'for' loop, i.e., at end of 3rd call to procedure Inseq:




At end of 'while' loop in procedure Inseq during 4th pass through 'for' loop in main program:



At end of 4th pass through 'for' loop, i.e., at end of 4th call to procedure Inseq:



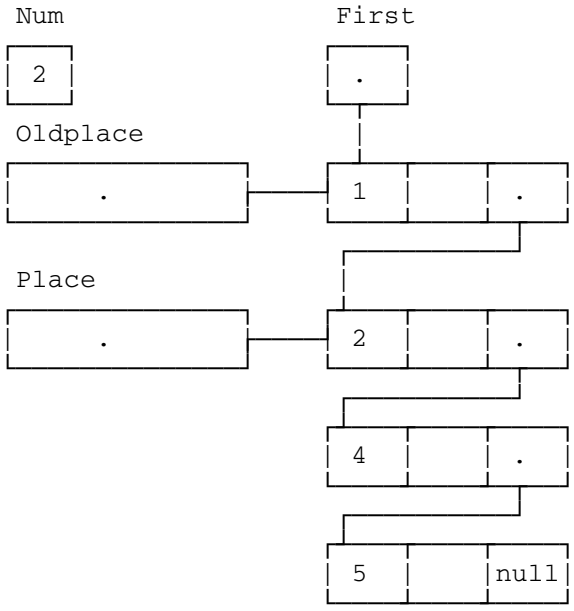

Deletion part

Recall that the data relevant to this portion of the program are:

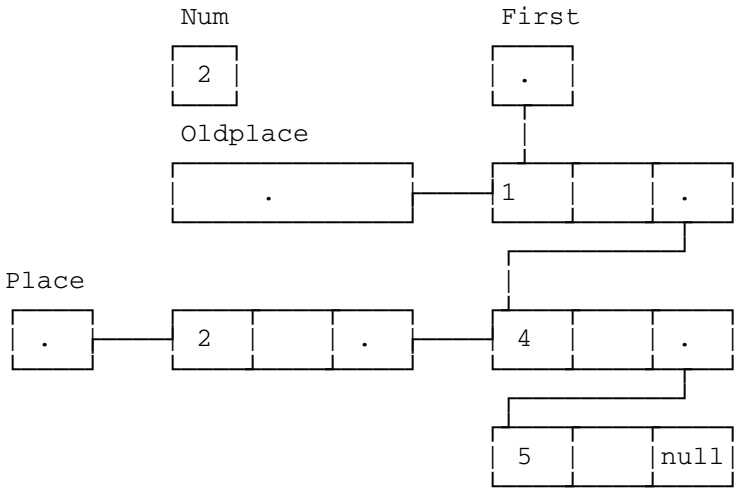
```
5
2
5
1
4
2
```

(1) Example of deletion in middle of list:

At end of 'while' loop in procedure Delete during 1st pass through 'for' loop in main program:



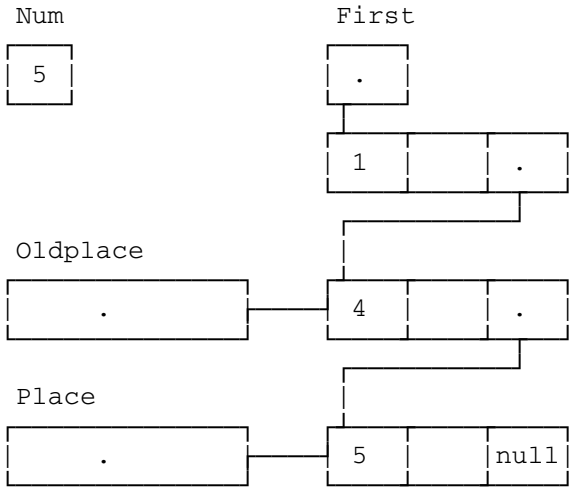

At end of 1st pass through 'for' loop, i.e., at end of 1st call to procedure Delete:



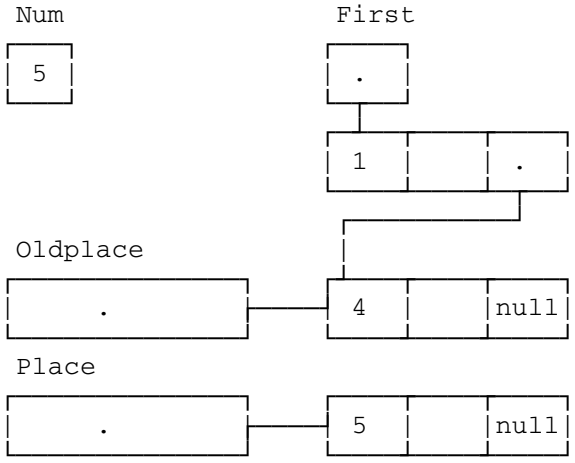

Record 2 is no longer in the list.

(2) Example of deletion at end of list:

At end of 'while' loop in procedure Delete during 2nd pass through 'for' loop in main program:



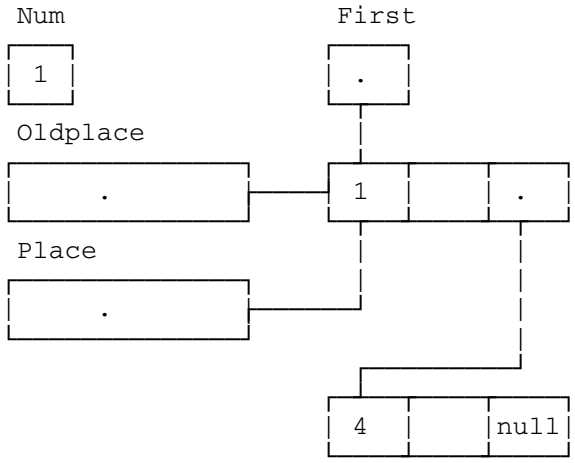

At end of 2nd pass through 'for' loop, i.e., at end of 2nd call to procedure Delete:



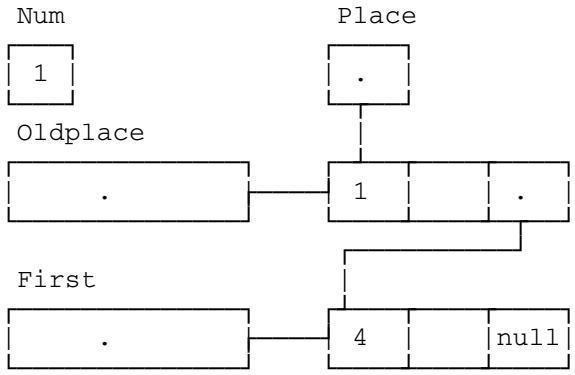

Record 5 is no longer in the list.

(3) Example of deletion at top of list:

At end of 'while' loop in procedure Delete during 3rd pass through 'for' loop in main program:

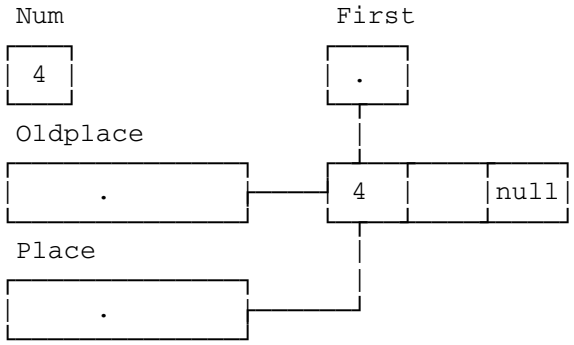



At end of 3rd pass through 'for' loop, i.e., at end of 3rd call to procedure Delete:

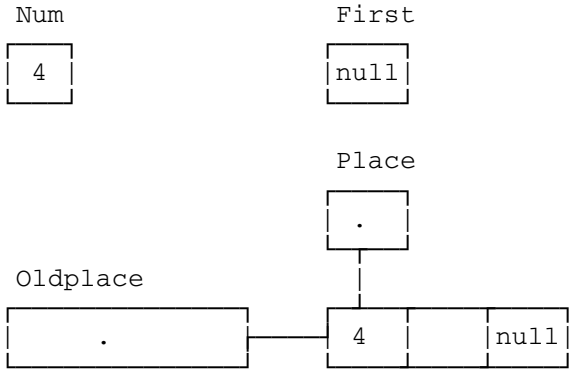



Record 1 is no longer in the list since no global variable points to it. Only local variables, Oldplace and Place, which change at the beginning of every pass, point to record 1.

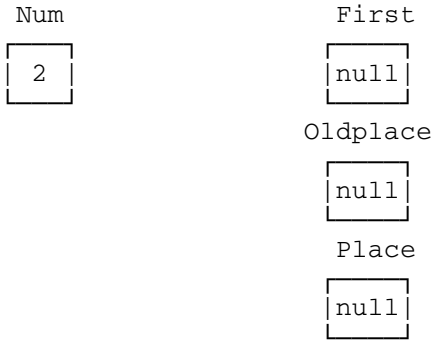
(4) At end of 'while' loop in procedure Delete during 4th pass through 'for' loop:




At end of 4th pass through 'for' loop, i.e., at end of 4th call to procedure Delete:




(5) At end of program:




Output: Nothing left in list

MULTIPLE RECORD CLASS DECLARATIONS

As indicated by the <record-class-identifiers> section in a reference declaration, it is possible for a reference variable to point to an occurrence of any of several record classes. This means that at any given time, a reference variable may have a value which points to a record occurrence of one record class, and at another time, the same variable may have a value pointing to a record occurrence of a different record class.

In certain situations, it may be desirable to find out to which record class a reference is currently pointing. A logical expression of the following form is used:

<reference-variable> is <record-class-identifier>

where:

<reference-variable> is an identifier, a subscripted variable or a field designator of type reference;

<record-class-identifier> is an identifier previously declared a record class and

'is' is a logical operator with the same precedence as the relational operators =, ¬=, etc. (see "Precedence" in the section "Logicals").

Note that reference variable must have been declared such that it could be pointing to the <record-class-identifier>.

The value of the logical expression is true if, and only if, the reference is pointing to a record occurrence belonging to the specified record class. It is false if the value of the reference is 'null' or if pointing to a record occurrence of a different record class.

A sample program using multiple record-class declarations and the operator 'is' is:

```
comment
   This program creates one linked list containing two different
   record types: Adult and Child. Values for the fields of each
   record occurrence are read and, depending on the value of
   age, either an Adult or a Child record occurrence created.
   Because different fields are required: Soc_Sec_No for adults
   and Guardian_Name for children, two record types are needed.
   Later, the list is processed from beginning to end and a
   different output printed for Adult and Child record types.
   The operator 'is' is used to determine to which record type a
   given record occurrence belongs.  ;

begin
```

```
integer Age,  Socsecno, N;
string(10) Name, Guardian;
comment
   Both record types have reference fields which can
   point to occurrences of either record type. However,
   Anext can only point from Adult records and Cnext only
   from Child records ;

record Adult(string(10) Aname; integer Soc_Sec_No;
             reference(Adult, Child) Anext);
record Child(string(10) Cname; string(10) Guardian_Name;
             reference(Adult, Child) Cnext);
reference(Adult, Child) Current, First;
comment N is the total number of record occurrences to be
created;

Read(N);
Read(Age);
if Age < 12
comment
   Keep track of the first record occurrence in the
   list. Create either a Child or an Adult record
   occurrence, depending on the age ;
then
begin
   Readon(Name, Guardian);
   First := Child(Name, Guardian, null)
end
else
begin
   Readon(Name, Socsecno);
   First := Adult(Name, Socsecno, null)
end;

comment
   Current is used to keep track of the record
   occurrence being processed currently ;
Current := First;
comment Build the linked list ;
for I := 1 until N - 1 do
begin
   Read(Age);
   comment
      If the current record occurrence is an Adult, then
      Anext points to the next record occurrence in the
      list. If it is a Child, Cnext points to the next
      record occurrence ;

   if Current is Adult then
   begin
      if Age < 12 then
      begin
         Readon(Name, Guardian);
```

```
               Anext(Current) := Child(Name, Guardian, null);
               Current := Anext(Current)
            end
            else
            begin
               Readon(Name, Socsecno);
               Anext(Current) := Adult(Name, Socsecno, null);
               Current := Anext(Current)
            end
         end
         else
         begin
            if Age < 12 then
            begin
               Readon(Name, Guardian);
               Cnext(Current) := Child(Name, Guardian, null);
               Current := Cnext(Current)
            end
            else
            begin
               Readon(Name, Socsecno);
               Cnext(Current)  := Adult(Name, Socsecno, null);
               Current := Cnext(Current)
            end
         end
      end;

      comment Start at the beginning of the list ;
      Current := First;

      comment
         While statement goes from beginning to end of list, using
         'is' to check for Adult or Child record types and
         printing appropriate outputs ;

      while Current ¬= null do
      begin
         if Current is Adult then
         begin
            Write(Aname(Current), "Adult dosage prescribed");
            Current := Anext(Current)
         end
         else
         begin
            Write(Cname(Current), "Child dosage prescribed");
            Current := Cnext(Current)
         end
      end
   end.
```

The following data:

```
 5
 25 "Benson"     145627431
 30 "Carp"       256483928
 8  "Dole"       "Dole"
 40 "Hahn"       456836672
 10 "Salter"     "Smith"
```

produce the resulting output:

```
Benson    Adult dosage prescribed
Carp      Adult dosage prescribed
Dole      Child dosage prescribed
Hahn      Adult dosage prescribed
Salter    Child dosage prescribed
```

# BASIC INPUT AND OUTPUT

Algol W provides several predeclared procedures to deal with input and output. All of these are accessible to any part of an Algol W program. One can imagine a "supplied" block containing all predeclared procedures. Since the user's program is (conceptually) inserted in this block after the declarations, all such procedures are global to the entire program.

All of the procedures described in this section assume that Algol W programs run in an environment which has a single input stream and a single output stream. The input stream is known to Algol W as INPUT and corresponds to the MTS logical I/O unit SCARDS. The output stream is known within Algol W as PRINT and corresponds to the MTS logical I/O unit SPRINT.

In fact Algol W programs run in an environment in which there are many input and output streams. Additional predeclared procedures, which are described in subsequent sections, allow Algol W programs to reference all of these streams either by the free format methods described in this section or by interpreting string format fields in a similar manner to that used by Fortran.

For most simple programs, the procedures described in this section will be sufficient when a single stream of data is being read in and a single set of results are to be sent to a printer.

## INPUT DATA

Data to be read by an Algol W program can appear in any of the first 256 columns of a data record. Data appearing after column 256 is ignored. Each data element is separated by one or more blanks or a comma from the next one. These blanks or commas are known as break characters. An input record (that is a line in a file or a card in a batch job) is considered to have 256 characters. If the physical line read in contains less than 256 characters, then it is padded on the right by blanks to a length of 256. Input records are read in sequence, as many as necessary to fulfill the requirements of the input statement. Data items input by the free format procedures described in this section must appear completely on one input record; such data items cannot be continued from one record to the next. Apart from this one restriction, the input records are viewed as being next to each other in one continuous stream.

Input data items may be written as constants of the appropriate types. The exception is that the trailing "L" of a long precision floating point quantity is never legal on an input data item. All floating point quantities read in by Algol W are converted initially to approximately the full double precision form. If data items specify a short precision floating point simple type, that is real or complex, then the value is truncated to short precision on assignment to the receiving variable.

Many additional forms are allowed for input data items for the convenience of Algol W programmers. Those for each simple type are described separately in the following subsections.

## Integer

Integer data items are normally supplied as optionally signed integer constants.

They may also be supplied as a bits constant, that is as a hash mark (#) followed by 1 to 8 hexadecimal digits. In this case the assignment to the receiving variable is of the integer equivalent of the bits quantity specified. The effect is as if the Number predeclared function had been called with the data item as its argument.

Examples:

```
0
1234
+99
-1001
#A9   ...... (equivalent to integer constant 169)
```

The first four items are valid integer constants within an Algol W program. All five are valid as input data items and supply a 32-bit integer value.

## Real and Long Real

When floating point quantities are converted to their internal form during an input operation, the conversion is always done to approximately full double precision. For this reason, there is no need for an input data item to specify long precision by a trailing "L" as would be necessary when specifying a long real constant in Algol W program text. An optionally signed real constant is a valid input data item for assignment to either a real or a long real variable. This real constant may however specify up to 17 significant digits. A long real constant (as specified by a trailing "L") is not legal as a floating point input item.

When a floating point quantity is being read into a real variable the double precision internal value is truncated to a short precision one before assignment to the variable. Any additional accuracy specified in the input data item is simply lost at this point. Short precision (real) variables can represent approximately 7 significant decimal figures; for long precision (long real) approximately 17 significant decimal figures may be held.

For convenience certain variations of the rigid format of a real constant are allowed for input data items. The exponent separator character, which must be a prime (') when specified in program text, is relaxed to include any of the following characters:

```
'  E  e  D  d
```

Examples:

```
0
-1.234
'5
1.36'-34
-3E7
+9.7d-8
```

The first four items above are valid real constants within Algol W program text where they would specify a number to short precision accuracy only. As input data items all six are valid and specify numbers to long precision accuracy.

## Complex and Long Complex

There are two forms allowed for data items to be input into complex or long complex variables.

The first form is similar to complex constants which may be specified within program text, but with the same changes and relaxations as specified in the previous section for real and long real data items.

In this form a complex number consists of an optionally signed floating point number (specifying the real or long real part) followed immediately by a signed floating point number suffixed by the letter "I" (specifying the imaginary or long imaginary part). No spaces may appear within this sequence. The presence of a trailing "L" on any floating point constant is illegal and the trailing "I" on the imaginary part may be specified in either upper or lower case.

If the imaginary part of the complex quantity is zero, then the signed floating point number specifying the imaginary part may be omitted together with its trailing I. If the real part of the quantity is zero then the number may be specified by a single optionally signed floating point number followed by a suffix of "I" or "i".

Examples:

```
0
-2.3
4.9I
3+4I
4.5'5+8.9'6i
6.9e-3-8.45e-4i
```

The first five values are valid complex constants within Algol W program text. All six values are valid complex or long complex data items and specify quantities to approximately full long precision. In the first two items the imaginary part of the number is zero. In the first and third items the real part of the number is zero. Note that if both parts of the number are zero, zero must still be given.

The second form is provided so that complex quantities output from an Algol W program may be subsequently re-input to another Algol W program.

In this form the real and imaginary parts of the number are specified as two optionally signed floating point data items, separated by a comma and with the whole group enclosed in parentheses. If the imaginary part of the number is zero it may be omitted. The real part must always be specified. Within the parentheses any number of spaces may be given provided that they do not occur within one of the floating point data items. Note that the imaginary part does not have a trailing "I" and that this character is illegal in this context.

Examples:

The examples given after the description of the first format may be written in the second format as:

```
(0, 0)
(-2.3, 0)  or:  (-2.3)
(0, 4.9)
(3,4)
(4.5'5, 8.9'6)
(6.9e-3, -8.45e-4)
```

Note that, while all of the numbers given in this second set of examples are valid complex or long complex data items, none of them are valid as complex constants within Algol W program text. This form is very similar to the format in which Algol W outputs complex floating point quantities.

As with real variables, the two halves of a complex data item are converted on input to approximately the full double precision accuracy. When the value is assigned to a receiving complex variable both the real and imaginary parts are truncated to short precision. Extra significant figures are simply lost at this point.

Of course with a long complex receiving variable no such truncation will occur.

## Logical

There are two valid logical constants within Algol W program text. These are:

```
true
false
```

Both of these constants, in any mixture of upper and lower case, are valid logical data items. For convenience any abbreviation down to "T" or "F" is also a valid data item. Any combination of mixed upper and lower case characters is valid.

Examples:

```
TRUE
False
T
F
tr
fa
```

Only the first two values are valid logical constants within Algol W program text. However all six are valid logical data items.

## Strings

Within Algol W program text, a string constant may be specified as a sequence of characters enclosed in quotes ("). If any quote occurs as a character within the string then two quotes must be specified. Such constants are valid data items for input into a variable of simple type string. Note however that the data item must be entirely on one physical input record: it may not be broken across a record boundary.

For convenience two other forms of string constants are allowed when the constant is an input data item.

In the second form the quote delimiters of a normal string constant are replaced by the prime (’). Note that in this form a quote as a character is not doubled but a prime must be. Primes in string constants which are delimited by quotes are not doubled.

The third form allows strings to be input without either quotes or primes as delimiters. In this form the string to be input may not

contain any embedded break characters (spaces or commas). Furthermore the first character of the string may not be a quote or a prime if such a character occurs elsewhere in the line, that is outside the bounds of the string being input.

Examples:

```
"JOHN BULL"
""""
'There''s an east wind coming, Watson.'
'"XX"'
Watson
Sherlock Holmes
```

Of the six data items given above, the first two are valid string constants within Algol W program text. Note that the second example sends only one character, a quote ("), but the fourth sends four characters because, in this case, the string delimiters are primes. In the sixth example the string sent is "Sherlock"; "Holmes" is the next data item because it is separated from the first string by a break character, the space.

The following rules apply to string data items:

(1) If the characters to be input into the string variable include break characters, that is spaces or commas, then the characters must be surrounded by delimiters. These are either quotes (") or primes ('). A string must be terminated by the same delimiter as that used to start it.

(2) If a character string delimited by quotes contains a quote character, then for this two quotes must be given.

(3) If a character string delimited by primes contains a prime character, then for this two primes must be given.

(4) If the character string to be input contains a quote or a prime as its first character and the same character occurs later in the same input record, then the string should be surrounded by delimiters.

(5) If the string to be input includes neither break characters nor a quote or a prime as its first character, then the input data item need not be surrounded by delimiters. In this case characters will be input from the first non-break character to the following break character or the end of the current physical data record, whichever occurs sooner.

(6) When a string is input between two delimiters, these are stripped from the characters as they are input and any double delimiters within the string are replaced by the equivalent single one.

(7) If the number of characters input into a receiving variable is greater than the length of the variable then a fatal error condition is recognized.

(8) If the number of characters input into a receiving variable is less than the length of the variable then the data item is padded on the right with blanks.

These rules reflect the default behavior of the string recognition algorithm used by Algol W. The rules may either be changed to remove recognition of primes or quotes as delimiters, or extended to recognize parenthesized groups, by suitable calls to the predeclared procedure Iocontrol. These facilities are described in "Modification of the String Recognition Algorithm" in the section "Miscellaneous Topics."

## Bits

Any bits constant valid within Algol W program text is also valid as a bits input data item. It is composed of a hash mark (#) followed by 1 to 8 hexadecimal digits. If less than 8 digits are given then the constant is assumed to be padded on the left by hexadecimal zeros. The hexadecimal digits A to F may be given in any mixture of upper and lower case characters.

For convenience, when a bits constant is specified as an input data item, the hash mark may be omitted.

Examples:

```
#0
#1234
#FF
#c1c2c3d4
abc
100
```

Of these six bits constants, only the first four are valid within Algol W program text. All six are valid as bits input data items. Note carefully that the final item specifies three hexadecimal digits, that is it will translate to an internal bits value of #00000100, not integer 100.

## Reference

Reference variables may not be assigned through an input statement. To assign a value to a record through an input statement, the individual field designators must be specified and, even so, reference pointers may not be read in.

## INPUT STATEMENTS

The following predeclared input procedures are available:

Read
Readon
Readcard

Input statements take the following form:

<procedure-identifier> (<input-parameter-list>)

where:

<procedure-identifier> is Read, Readon or Readcard;

<input-parameter-list> is a list of <variables> and/or <simple-statements>, each separated from the next by a comma;

<variable> is an identifier, a subscripted variable indicating an element of an array, a field designator indicating a field of a record occurrence or a substring designator indicating a portion of a string; and

<simple-statement> can be any simple statement but, in general, is a format assignment statement or a control statement.

Format assignment and control statements, although legal in input statements, are mainly relevant to output ones (see exception of Iocontrol("NEXTCARD") under "Control of Basic Input and Output" later in this section). Therefore the typical input statement is of the form:

<procedure-identifier> (<variables>)

### Read and Readon

Both Read and Readon are free format procedures. In other words, values on data records do not have to be spaced in any particular way, except for requiring one or more break characters to separate them. Values are read, matched with the variables in the input parameter list in order of appearance and assigned to the corresponding variables.

The type of each data item must be assignment compatible with that of the corresponding variable. The types of variables which may be read are: arithmetic (that is integer, real, long real, complex and long complex), string, bits and logical. Values of reference variables may not be read in as input data.

A Read statement begins reading at the beginning of a new input record, whereas a Readon begins reading on the same input record as the previous Read or Readon statement. Both Read and Readon continue reading values until all variables in the input parameter list have been matched. If necessary, either predeclared procedure will read in more than one physical input record to satisfy the request. If more variables are listed in the input statement than there are values on the data records, then an error condition is recognized.

Examples:

```
begin
   real A, B;
   integer I, J;
   Read(A, I);
   Read(B, J)
end.
```

If the data records are as follows:

```
-4.2    6
 5.1    -19
```

A has the value -4.2, I the value 6, B the value 5.1 and J the value -19.

```
begin
   real A, B;
   integer I, J;
   Read(A, I, B);
   Readon(J)
end.
```

If the data records are as follows:

```
-8.9    9
1.5     -32
```

A has the value -8.9, I the value 9, B the value 1.5 and J the value -32. The Read statement reads two values from the first input record and goes to the second one for the third value.

```
begin
   complex C;
   integer I, J, K;
   real A, B;
   Read(C, A, K);
   Readon(B, I);
   Read(J)
end.
```

If the data input records are as follows:

```
-4.2+3I   60.4
-15       -4.3'8
49   64
28
```

C receives the value -4.2+3I, A the value 60.4, K the value -15, B the value -4.3'8, I is 49 and J is 28. The 64 is ignored.

Note that a data item input by a Read or a Readon statement may not be broken across two physical input records. This applies to all data types including strings.

## Readcard

Readcard designates a procedure which reads all the characters of an input record into a corresponding string variable. The data characters read by a Readcard statement are not enclosed by quotes or primes unless these characters are to be input. A Readcard statement causes reading to start at the beginning of a new input record. Any subsequent input statement begins at the first character of the following record. The string variable may be of any length. If a string variable listed in the Readcard input parameter list is longer than the number of characters input, the remaining characters of the variable are padded with blanks. If a string variable is shorter than the number of characters input, then the input record is truncated on the right and just those characters which will fit are transferred to the variable.

Note that all variables in the Readcard input parameter list must be string ones. They may be entire strings or substring designators.

Examples:

```
begin
   string(24) Text;  string(100) Line;
   Readcard(Text, Line(10|85))
end.
```

Text now contains the string value of the first 24 characters of the first data input record. Line has the following composition: characters 1 to 10 are undefined, 11 to 95 contain the string value of the first 85 characters of the second data input record and 96 to 100 are undefined.

```
begin
   string(80) Line, Text, Word;
   Readcard(Line, Text);
   Read(Word);
   Write(Line);
   Write(Text);
   Write(Word)
end.
```

If the data lines are:

```
"Quotes are not always needed"
Quotes are not always needed
"Quotes are not always needed"
```

the output is:

```
"Quotes are not always needed"
Quotes are not always needed
Quotes are not always needed
```

## OUTPUT STATEMENTS

Algol W provides the following predeclared output procedures:

```
Write
Writeon
Writecard
```

Output statements take the following form:

```
<procedure-identifier> (<output-parameter-list>)
```

where:

<procedure-identifier> is either Write, Writeon or Writecard;

<output-parameter-list> is a list of expressions and/or <simple-statements>, each separated from the next by a comma;

<expression> is any legal expression; and

<simple-statement> can be any legal simple statement but, in general, is a format assignment statement or a control statement (see "Format Specifications and Assignment Statements" later in this section).

### Write and Writeon

Format conversion for Write and Writeon is performed automatically. In other words, no statements equivalent to Fortran format statements are required. The values of the variables in the output parameter list are printed in succession, in order corresponding to their order in the list. Output records (that is lines on the printer or lines in an output file) are up to 256 characters in length. However, for the default output stream used by Write and Writeon (that is PRINT, which is

equivalent to MTS SPRINT) the default output length is 133 characters. If the value of an output expression cannot fit on the current line, it is printed on the next one. The default output length of the stream in use may be changed under program control using the Qualify predeclared procedure which is described in the next section. Values of expressions listed in a Write statement are output starting from the first character of a new line. Values of expressions listed in a Writeon statement are printed on the same line as the last value printed by the previous Write or Writeon statement. If the value of an expression listed in a Writeon statement will not fit on the current output record then a new output record is started.

Example:

```
begin
   real X, Y, Z;
   integer I, J;
   Read(X, Y, I, Z, J);
   Write(I, Z);
   Writeon(X, J);
   Write(Y)
end.
```

If the data are as follows:

```
5.2  6.4  -8      4.8 +20
```

the output appears as follows:

```
      -8        4.800000        5.200000             20
6.400000
```

Note that constants may appear as part of the <output-parameter-list>.

Examples of legal output statements:

```
Write(X, " is the value of X")
Write(9, 10)
```

## Complex Expressions

Complex expressions are output in a form in which they could be re-input to a complex variable. In order that the real and imaginary parts may appear in columns when output data is tabulated, a complex expression is output in the form:

```
(x,y)
```

where x is a real output field displaying the real part of the complex expression and y is a real output field displaying the imaginary part. The space taken up on the output data record is a field consisting of a left parenthesis followed by a floating point real field, a comma, the floating point imaginary field and a right parenthesis.

Example:

```
begin
   complex P, Q;
   P := 3.4+5.6I;
   Q := -0.3+67.5I;
   Write(P);
   Write(Q)
end.
```

Using ␣ to represent blank, the output from this program is:

```
(␣␣␣␣3.400000␣,␣␣␣␣5.600000␣)
(␣␣␣␣-0.3000000,␣␣␣␣67.50000␣␣)
```

To output a complex quantity without parentheses, the real and imaginary parts of the number must be output separately using the relevant predeclared functions.

Example:

```
begin
   complex P, Q;
   P := 3.4+5.6I;
   Q := -0.3+67.5I;
   Write(Realpart(P), Imagpart(P));
   Write(Realpart(Q), Imagpart(Q))
end.
```

The output from this program is:

```
␣␣␣␣3.400000␣␣␣␣␣␣␣␣5.600000␣
␣␣␣␣-0.3000000␣␣␣␣␣␣67.50000␣␣
```

String Expressions

String expressions are output in a field length exactly equal to that of the string. However, whereas all other expressions in Algol W are output completely on one record, in the case of a string this is not true. Output of characters from the string starts on the current output record then, if the string will not fit completely onto this, the characters of the string are broken over as many further records as are necessary.

## Writecard

Writecard designates a procedure which writes the whole of the supplied string argument on a single output record. Each Writecard argument starts a new output record and any subsequent output by any output procedure will also start a new record.

If a string expression output by Writecard is longer than the maximum length of an output record for the basic output stream then it is truncated on the right.

Note that all expressions in the Writecard parameter list must be of simple type string. They may be entire strings or substring designators.

Example:

```
begin
   string(48) Text;
   Text := "Mary had a little lamb";
   Writecard(Text, "Whose fleece was white as snow")
end.
```

The output from this program is :

```
Mary had a little lamb
Whose fleece was white as snow
```

With Write and Writeon statements, records are only written out when the current output record buffer is full. In the case of Writecard, the buffer is assumed to be full after each argument and its contents are forced out immediately.

# FORMAT SPECIFICATIONS AND ASSIGNMENT STATEMENTS

## Format Variables

Output formats are controlled by predeclared variables. All of these are initialized to default values which can be changed by the use of format assignment statements. Both Write and Writeon are procedures which output the values of the expressions listed in the output parameter list according to the formats specified for the given types. No format control applies to the Writecard predeclared procedure.

An alternative to using these predeclared variables is to use the formatted I/O statements described in the section "Format Directed Input and Output." Using the formatted I/O statements is generally preferred over using the predeclared format control variables.

The following predeclared variables control the formats of the various types:

I_W

I_W is a variable of type integer which sets the width of integer fields on output. Its value is initialized to 14 but can be changed in a format assignment statement such as:

```
I_W := 6
```

R_W

R_W is a variable of type integer which sets the width of real and long real fields. Its value is initialized to 14 but can be changed in a format assignment statement. Because complex and long complex numbers are composed of two reals and two long reals, respectively, R_W controls the widths of these types indirectly. Their widths are 2*R_W + 3 and thus have the initial value of 31.

R_D

R_D is a variable of type integer which sets the number of places after the decimal point in real, long real, complex and long complex fields of output. Its initial value is 0 but can be changed with a format assignment statement such as:

```
R_D := 4
```

Note that the R_D variable is relevant only if the R_Format value is set to "F" or "A" - see R_Format below.

R_Sig

R_Sig is a variable of type integer which sets the minimum number of significant digits which may appear in real, long real, complex and long complex fields of output. Its initial value is 3 but can be changed with a format assignment statement such as:

```
R_Sig := 5
```

Note that the R_Sig variable is relevant only if the R_Format value is set to the default of "G" - see R_Format below.

R_Expchar

R_Expchar is a variable of type string(1) which defines the exponent separator character which will be used in real, long real, complex and long complex fields of output. It is initially set to be the prime (') character but can be changed with a format assignment statement such as:

```
R_Expchar := "E"
```

Note that the R_Expchar variable is relevant only if the number is being output in explicit exponent form - see below.

R_Format

R_Format is a variable of type string(1) which controls the format of real, long real, complex and long complex values. The possible values for R_Format are:

| R_Format | Meaning | Fortran Equivalent |
|---|---|---|
| "G" | General (default) | None |
| "F" | Fixed decimal point | F |
| "E" | Explicit exponent | E or D |
| "D" | Same meaning as "E" | E or D |
| "A" | Same meaning as "F" | F |
| "S" | Same meaning as "E" | E or D |

The lower case equivalents of these characters may be specified if so desired.

S_W

S_W is a variable of type integer which sets the number of blanks to be added at the end of each field of output for all data types except strings. Its initial value is 2, which means that two blanks are inserted between adjacent fields on output unless the user resets S_W with a format assignment statement. String fields are always as long as the string itself and no blanks are appended.

## Fixed Decimal Point Format

Fixed decimal point format specifies that the current R_W value is the entire width of the value to be printed and that the current R_D value is the number of places to the right of the decimal point. The format used is:

```
x.y
```

where "x" is a signed integer representing the integral part and "y" is the specified number of digits representing the fractional part. The scale factor, indicated by the prime (') of the Algol W notation, is not used.

Example:

```
begin
   real A, B;
   R_D := 4;
   R_Format := "F";
   A := 8.40;
   B := -12.56139;
   Write(A, B)
end.
```

This program produces the output:

```
␣␣␣␣␣␣␣␣8.4000␣␣␣␣␣␣␣␣-12.5614␣␣
```

where each "␣" represents one blank. The first value has two 0's appended to it to meet the requirement of R_D being set to 4. Eight blanks precede it in order for the entire width to total 14, the default value of R_W. Eight blanks occur between the two numbers. The first two are a result of the default S_W setting. The next six are to supplement the width of the value of B in order to have a field width of 14. Because R_D is set to 4, the number has to be rounded from 5 decimal places to 4.

## Explicit Exponent Format

Explicit exponent format specifies that the Algol W prime (') notation is to be used. The number of significant digits is 7 less than the current R_W value. Thus the R_D value is irrelevant if R_Format is set for the explicit exponent form.

The format of real values on output is as follows:

```
x'y
```

where "y" is a signed, two-digit integer representing the power of 10 by which "x" (the mantissa) should be multiplied and "x" is an "unscaled" real, meaning:

```
1.0 <= x < 10.0
```

The exponent separator character, which defaults to a prime, is taken from the value of a predeclared string(1) variable, R_Expchar. If any other exponent separator character is desired in output, it can be obtained by assigning a new value to this variable.

Example:

```
begin
   real A, B, C;
   R_Format := "E";
   R_W := 13;
   A := 5.08;
   B := 53.4'2;
   C := -4976.253;
   Write(A, B);
   R_Expchar := "E";
   Write(C)
end.
```

The output from this program is:

```
␣␣5.08000'+00␣␣␣␣5.34000'+03␣␣
␣-4.97625E+03␣␣
```

Note that in this format a positive mantissa is preceded by two blanks and a negative one by only one.

## General (Default) Format

General format always attempts to print a value in a reasonable format as determined by the magnitude of the number and the number of significant digits desired.

Algol W first attempts to output the number in fixed point format in a field which is R_W characters long. The position of the decimal point is determined by integer dividing R_W by two. This value is used to provide the number of digits after the decimal point. The value of R_D is ignored in this format.

If the number will not fit into the supplied field width in fixed point format, the number will be output in explicit exponent form using the same value of R_W. The number will not fit if either its magnitude is too large, so that there is insufficient room for the digits preceding the decimal point, or insufficient significant digits would be printed.

The number of significant digits which must be printed in this format is determined by the value of the predeclared integer variable R_Sig, which defaults to three. This means that if a number can be output in fixed point format but less than R_Sig significant digits would be displayed, the number is printed in explicit exponent form instead.

A maximum of 7 significant digits will be printed for short precision quantities and 15 significant digits for long precision ones. If the last significant digit is reached before the decimal point, the number

is padded with zeros up to the decimal point and spaces are printed there after up to the end of the field. If the last significant digit occurs after the decimal point then the subsequent character positions again contain spaces.

Example:

```
begin
   real A;
   R_Sig := 5;
   A := 8.4;          Write(A);
   A := 12.56139      Write(A);
   A := 0.00034567;   Write(A);
   A := 45.5'8;       Write(A)
end.
```

The output from this program is:

```
␣␣␣␣8.400000␣␣␣
␣␣␣12.56139␣␣␣␣
␣␣3.456700'-04␣␣
␣␣4.550004'+09␣␣
```

The program does not make any assignment to R_Format or R_W. The default values of "G" and 14 respectively are used. The number of significant digits set by R_Sig is changed to specify a minimum of 5. Note that the first two numbers are printed in fixed point format with all 7 significant digits displayed. If the third number was displayed in this format, only four significant digits could be printed since Algol W attempts to keep the decimal point aligned for columnar output. For this reason, it is output in explicit exponent form. In the case of the final number, there is insufficient room before the decimal point to display the leading significant digits so once again the number is printed in explicit exponent form.

## Simple Variable Types and Output Formats

The following table summarizes the predeclared format variables and their default values:

| Identifier | Type and Default | | Interpretation |
|---|---|---|---|
| I_W | integer | 14 | width of integer fields |
| R_W | integer | 14 | width of real and long real fields; width of complex and long complex fields (2*R_W + 3) |
| R_D | integer | 0 | places following the decimal point in real, long real, complex and long complex fields in fixed point format |
| R_Expchar | string(1) | "'" | the exponent separator character used when a real, long real, complex or long complex expression is output in explicit exponent form |
| R_Sig | integer | 3 | the minimum number of significant digits to be displayed in the default general format |
| R_Format | string(1) | "G" | format of real, long real, complex and long complex fields |
| S_W | integer | 2 | width of the fields of blanks appended to the end of each field (excluding string fields) |

Any values assigned to I_W, R_W or S_W in excess of 32 are treated as 32. If values of I_W, R_W, R_D, R_Sig or R_Format are inconsistent with the magnitude or precision of the numbers to be printed, alternative format values are chosen. These alternate choices ensure that an approximation to the number is given, and that no more digits are printed than are warranted by the precision of the number. Specifically, this means that if I_W is too small its size is disregarded and the integer is printed; rounding occurs if R_D is too small and the "F" value of R_Format is overlooked if the magnitude of the number to be printed is too great for the current R_W value.

Example:

```
begin
   integer I, J;
   Read(I, J);
   I_W := 1;
   Write(I, J)
end.
```

If the data card reads:

```
4802   9
```

the output is:

```
4802¢¢9¢¢
```

as if I_W had been increased temporarily from 1 to 4 to accommodate the larger value of the output expression I.

The following table explains the distribution of blanks and the field lengths for output expressions according to type of expression:

| Type | Field-Description |
|---|---|
| integer | Right-justified in a field of I_W characters and followed by S_W blanks |
| real and long real | Right-justified in a field of R_W characters and followed by S_W blanks |
| complex and long complex | Right-justified in two fields each of R_W characters separated by a comma, enclosed within parentheses and the whole followed by S_W blanks |
| logical | Left-justified in a field of 5 characters and followed by S_W blanks |
| string | Field length is exactly the length of the string |
| bits | Right-justified in a field of 9 characters and followed by S_W blanks |

## Format Assignment Statements

As mentioned earlier, format assignment statements may appear within an input or output statement as well as outside them.

Examples:

```
Write(I_W := 3, I, I_W := 4, J)
Write(I, I_W := 5, J, K)
```

These output statements contain format assignment statements to affect the output of the variables within them.

When a format variable is changed within an input/output statement, it is considered a local variable. In other words, the change has effect only while executing the current input/output statement. At the end of such a statement, the current value of the global variable is restored. Format assignment statements outside of input/output statements are global, and affect all subsequent output, until another format assignment statement changes the value of the format variable.

## IOCONTROL

Algol W provides the Iocontrol predeclared procedure to modify the action of the input/output system.

The form of an Iocontrol statement is:

Iocontrol(<control-parameter-list>)

where:

<control-parameter-list> is a list of <expressions> each separated from the next by a comma; and

<expression> is any legal expression.

In practice, the expressions should be either integer or string ones. Those of other simple types are ignored. The following table gives a list of all of the expressions which are recognized by Iocontrol. If an integer key is given which is not in the list, it is ignored. If a string expression is given, then it may contain one or more of the keywords specified, separated by commas. Keywords may be unambiguously abbreviated down to three characters and specified in any mixture of upper and lower case. If a keyword is given which is either not in the list below or not an unambiguous abbreviation of a list item, then Algol W will treat this as a fatal error.

It is strongly recommended that the string keywords be given as arguments to Iocontrol rather than using the equivalent integer values, as this is less prone to error and enhances source program readability.

Note that just like format assignment statements, Iocontrol statements may be inserted in Read or Write statements.

The following input/output control keywords modify the action of the basic input/output predeclared procedures previously described in this

section. Their action is fully described in the subsection "Control of Basic Input and Output" later in this section.

| Integer | String Keyword |
|---|---|
| 1 | NEXTCARD |
| 2 | SPACE |
| 3 | EJECT |
| 4 | NORMAL |
| 5 | FULLPAGE |
| 6 | DOUBLESKIP |
| 7 | TRIPLESKIP |
| 8 | OVERPRINT |

The following input/output control keywords control the timing information which is printed at the end of an Algol W program execution. They are described in the subsection "Timing Information" in the section "Miscellaneous Topics."

| Integer | String Keyword |
|---|---|
| 101 | TOTALCPU |
| 102 | PROBLEMCPU |
| 103 | SUPERCPU |
| 104 | ELAPSED |
| 105 | NOTIMES |
| 106 | ALLTIMES |

The following input/output control keywords control the action which the Algol W runtime error procedures will take if a program interrupt occurs while program execution is in either a non-Algol W external subroutine or an Algol W library routine. They are described in the subsection "External and Library Interruptions" in the section "Miscellaneous Topics."

| Integer | String Keyword |
|---|---|
| 201 | DISPLAY |
| 202 | NODISPLAY |
| 203 | PSW |
| 204 | GRS |
| 205 | FRS |
| 206 | SYSPGNT |
| 207 | ALWPGNT |

The following input/output control keywords control the action of the Getstring predeclared procedure. Getstring is described in the section "Stream Directed Input and Output." The effect of the various Iocontrol keywords on Getstring is described in the subsection "Control of Getstring Action" in the section "Miscellaneous Topics."

| Integer | String Keyword |
|---|---|
| 301 | GSFIELDED |
| 302 | GSRETURNS |
| 303 | GSCONTINUE |
| 304 | GSORIGIN |

The following input/output control keywords control the action of the string recognition algorithm when strings are read in free format using the predeclared procedures Read, Readon, Get, Geton and Getstring. The predeclared procedures Get, Geton and Getstring are described in the section "Stream Directed Input and Output." The action of the other keywords is described in the subsection "Modification of the String Recognition Algorithm" in the section "Miscellaneous Topics."

| Integer | String Keyword |
|---|---|
| 401 | RESETSCAN |
| 402 | NOQUOTES |
| 403 | QUOTES |
| 404 | NOPRIMES |
| 405 | PRIMES |
| 406 | BRACKETS |
| 407 | NOBRACKETS |
| 408 | DELBRACKETS |

## Control of Basic Input and Output

When supplied as arguments to Iocontrol, the following keywords control the action of the basic input/output predeclared procedures described earlier in this section. The parenthesized numbers are the integer equivalants to the keyword arguments. Use of integer arguments is discouraged.

The following keyword affects input operations.

NEXTCARD (1)

NEXTCARD causes the current contents of the input buffer within Algol W to be forgotten. The next input operation will fetch a new physical input record regardless of whether the predeclared procedure used is Read, Readon or Readcard. Read and Readcard would normally fetch a new input record so:

```
Iocontrol("NEXTCARD");
```

would cause a subsequent Readon to fetch a new input record as if Read had been called. As an example consider the following program:

```
begin
   real array Data(1::24);
   for I := 1 until 24 do
   begin
      if (I rem 6) = 1 then Iocontrol("NEXTCARD");
      Readon(Data(I))
   end;
    .
    .
end.
```

The program above reads in the 24 elements of an array Data. The effect of the If statement is that a call of:

```
Iocontrol("NEXTCARD");
```

is issued before the Readon for values of I which are 1, 7, 13, and 19. This ensures that a new physical record is fetched initially and then after every sixth data item read in.

The remaining keywords described in this section affect only the output predeclared procedures Write, Writeon and Writecard. Five of these keywords form a precedence hierarchy. If more than one call to Iocontrol is made with no intervening call to an output predeclared procedure then, if two or more keywords specified in the list of five below are given, the effect is that of the highest in the list:

OVERPRINT
EJECT
TRIPLESKIP
DOUBLESKIP
SPACE

For instance in the case:

```
Iocontrol("EJECT"); Iocontrol("DOUBLE");
```

The "DOUBLE" keyword has no effect because "EJECT" is higher in the hierarchy.

All of the keywords described below cause the current contents of the output buffer to be forced out to the relevant device. They then set the carriage control character at the start of Algol W's internal output buffer to a value which corresponds to the action specified. None of the keywords actually starts a new output record. This is why two consecutive Iocontrol calls do not force out a blank line.

SPACE (2)

SPACE causes the current contents of the output buffer to be written out to the attached file or device. Any subsequent Write or Writeon statement will start a new output record with a space as a carriage control character in column one. This carriage control setting will be overridden if another call to Iocontrol specifies any other member of the hierarchy.

The effect of the SPACE keyword is therefore simply to force out the contents of the output buffer. For example the sequence:

```
Write("Enter data item");
Iocontrol("SPACE");
Read(Item);
```

causes the string "Enter data item" to be written out before the system prompt caused by the Read of Item. It is in fact exactly equivalent to:

```
Writecard("Enter data item");
Read(Item);
```

It is especially important to understand this behavior when writing programs which interact with the user at a conversational terminal.

EJECT (3)

This keyword is similar in action to SPACE except that the carriage control character of a subsequent line will be "1", causing a skip to a new page if the output is being directed to a printer. A call of Iocontrol with a keyword of OVERPRINT (see below) would override this action if it occurred before the next output procedure call.

FULLPAGE (4)

This keyword is obsolete and is documented only for completeness. The argument number 4 was allocated in previous versions of Algol W but it is no longer supported.

NORMAL (5)

This keyword is obsolete and is documented only for completeness. The argument number 5 was allocated in previous versions of Algol W but it is no longer supported.

DOUBLESKIP (6)

This keyword behaves as SPACE but the carriage control character of the new record will be "0", causing a double skip before printing. It may be overridden by a subsequent call to Iocontrol specifying TRIPLESKIP, EJECT or OVERPRINT as these are higher in the hierarchy.

TRIPLESKIP (7)

This keyword behaves as SPACE but the carriage control character of the new record will be "-", causing a triple skip before printing. It may be overridden by a subsequent call to Iocontrol specifying EJECT or OVERPRINT as these are higher in the hierarchy.

OVERPRINT (8)

This keyword behaves as SPACE but the carriage control character of the new record will be "+". The effect of this on a printer is to cause the subsequent line to be overstruck onto the characters of the previous one. A common use of this is to underline text.

Example:

```
string(80) Title;
 .
 .
Title := "Listing program output";
Iocontrol("EJECT");
Writecard(Title);
for I := 0 until 79 do
if Title(I|1) ¬= " " then Title(I|1) := "_";
Iocontrol("OVERPRINT");
Writecard(Title);
```

If this section of program executes with the output directed to a printer then the effect is to start a new page with the underlined text:

Listing program output

Special characters such as the Danish "ø" can be produced in this way.

NEWLINE

Iocontrol is a passive control procedure: its call does no input/output. It simply specifies the behavior of subsequent input/output operations. Consecutive calls to Iocontrol specifying a number of lines to be skipped on output, cause only one of those calls to be recognized if there are no intervening output statements.

For instance:

```
Write("ABC");
Iocontrol("SPACE");
Iocontrol("SPACE");
Write("DEF");
```

would cause the following to be printed:

```
ABC
DEF
```

Notice that the second call of Iocontrol does not cause an additional blank line to be skipped. The Algol W run time system has already been told by the first call that the subsequent output operation is to start in column one of a new physical output record. Telling it again has no additional effect.

The predeclared procedure Newline allows line skip control commands to be issued which will skip output lines and which take immediate effect.

The general form of the procedure call is:

```
Newline(<argument>)
```

where:

<argument> is either an <integer-expression> or a <string-expression>.

If the argument to Newline is given as an integer expression then this is used to determine the number of blank lines which will be written. If the integer expression evaluates to a number between 1 and 60 inclusive then that number, minus one, blank lines are produced and a new physical output record is started. If the expression evaluates to a number greater than 60 or less than zero then a run-time error occurs. If the value is zero then a single new line is started with a carriage control character of "+" causing the new line to overstrike the previous one.

If, on the other hand, the argument is given as a string expression, a single new line is started with the carriage control character equal to the first character of the supplied string expression.

All calls to Newline cause any partially built output record to be output. If the simple type of the argument to Newline is not integer or string then no further action takes place. Therefore:

```
Newline(null);
```

is exactly equivalent to:

```
Iocontrol("SPACE");
```

The following example shows one method by which a prompt line may be written in Algol W:

```
integer Num;
 .
 .
Newline(3);
Newline("&");
Writeon("Enter number");
Newline(null);
Read(Num);
```

In this example the first call of Newline causes three blank lines to be written. The second call causes a new output record to be started with an ampersand (&) as the carriage control character. This is the MTS logical carriage control for "write and do not carriage return." (Note that this particular carriage control character is valid only at conversational terminals.) The final call to Newline forces the prompt line to be written before the Read call prompts for data.

## CARRIAGE CONTROL CHARACTER GENERATION

Carriage control characters are generated by default in Algol W when the predeclared procedures Write, Writeon and Writecard are used to produce output. If desired, this action may be suppressed by use of:

(1) the run time parameter NOCC, described under "Run Time Parameters" in the section "Algol W Programmer's Guide";

(2) by assignment to a predeclared logical variable, Write_Cc.

If Write_Cc is set to 'false' then no carriage control characters are produced and the first character of each output record will be the first character specified in the relevant output statement.

Non-generation of carriage control characters does not prevent MTS from acting on a presumed carriage control directive. On devices such as printers and terminals, the first character of each output record will normally be stripped off and used as a carriage control instruction. Care must be taken to specify this first character correctly if this is the case.

Common carriage control characters and their meanings are:

| Character | Meaning |
|---|---|
| ⊬ø (blank) | skip to a new line |
| 0 | skip two lines |
| - | skip three lines |
| 1 | skip to a new page |
| + | overstrike previous record |

The above list gives American Standards Association (ASA) standard carriage control. MTS supports other carriage control characters - see Appendix H of the section "Files and Devices" in MTS Volume 1, The Michigan Terminal System. If there is any doubt about the requirement for a control character in a particular application, a blank character should be given as the first character of each output record. As previously explained, this is done by default for the predeclared procedures Write, Writeon and Writecard. Iocontrol and Newline assume the above list of carriage control characters when invoked.

## SAMPLE INPUT OUTPUT PROGRAM

```
comment The following program contains some examples of
        procedures for formatting output in Algol W;
begin
   comment The formal parameter N stands for the total
           number of significant digits desired: add 7 to
           to get field width R_W;

   procedure Scaled(integer value N);
   begin
      R_Format := "E";
      R_W := N + 7
   end;

   comment D stands for the number of digits to the
           right of the decimal point. N stands for those
           to the left.  The field width R_W is the sum
           of N+D+1, where 1 is for the decimal point;

   procedure Aligned(integer value N, D);
   begin
      R_Format := "F";
      R_W := N + D + 1;
      R_D := D
   end;

   comment General needs the same R_W value as Scaled;

   procedure General(integer value N);
   begin
      R_Format := "F";
      R_W := N + 7
   end;

   procedure Line_Skip;
      Iocontrol("SPACE");
```

```
        comment Main program starts here;
        General(5);
        I_W := 2;
        S_W := 1;
        for I := -1, 0, 32 do
        begin
           Write(S_W := 0, I, ":", Line_Skip, I/3);
           Writeon("I  ", Aligned(3,2), I/3,
              "*", Scaled(12), I/3, "*")
        end
     end.
```

The output from this program is:

```
        -1:
              -0.333I   -0.33 * -3.33333333333'-01 *
         0:
               0.000I    0.00 *  0.00000000000     *
        32:
              10.667I   10.67 *  1.06666666667'+01 *
```

# MULTIPLE INPUT AND OUTPUT STREAMS

Input and output streams are channels by which information can pass between a program and particular files or devices. In the last section, "Basic Input and Output," it was assumed that Algol W ran in an environment which has a single input stream and a single output stream. In fact there are many input/output streams available to an Algol W program. This and the following section describe how Algol W programs may use and control multiple input/output streams.

## INPUT/OUTPUT STREAMS AND STREAM DESIGNATORS

There are two kinds of input/output stream:

(1) Predefined input/output streams. Algol W knows about these before the program starts to run. There are 25 predefined input/output streams: 5 named streams and 20 numbered ones.

(2) User defined input/output streams. These may be added and deleted by a program as it runs by calling the predeclared procedures Assign and Release.

Associated with each stream is a stream designator, which may be used within an Algol W program to refer to that particular stream. Stream designators are sometimes also called stream names or stream numbers. Within an Algol W program, stream designators may be given as string expressions, integer expressions or special predeclared bits variables, depending on the type of stream being referenced.

Most streams also have an MTS file or logical I/O unit attached to them (see MTS Volume 1, "The Michigan Terminal System," for a description of MTS files and logical I/O units). When a program uses a particular stream to read or write data, that data is read from or written to the MTS file or unit attached to that stream. For example, the streams INPUT and OUTPUT (used by default by the basic input and output procedures Read, Write, and so on) are attached to the MTS logical units SCARDS and SPRINT, respectively. Thus these basic input and output procedures by default read from the file or device assigned to SCARDS and write to the file or device assigned to SPRINT.

## Predefined Named Input/Output Streams

The following table lists the 5 predefined named streams.

| Stream Name | Attached MTS I/O Unit | Purpose |
|---|---|---|
| INPUT | SCARDS | Main input data stream. |
| PRINT | SPRINT | Main printed output stream. |
| PUNCH | SPUNCH | Punched card output. |
| ERROR | SERCOM | Algol W diagnostic output. |
| USER | GUSER | User prompts by the system. |

By default the predeclared procedures Read, Readon and Readcard read data through INPUT and procedures Write, Writeon, Writecard, Iocontrol, and Newline write output records through PRINT.

A stream designator for a predefined named stream may be given in two ways: one, as any string expression which evaluates to the appropriate name (e.g. "PRINT"); or two, as a special predeclared bits variable which has the same name as the associated stream (e.g. Print).

As noted in the above table, each predefined named stream is attached to an MTS logical I/O unit (e.g. INPUT is attached to SCARDS). Note that the predefined named streams may not be attached to any other file or device from within an Algol W program.

Note also that the MTS logical I/O units SERCOM and GUSER should not normally be directed away from the default operating system assignments (*MSINK* and *MSOURCE*, respectively). Otherwise important Algol W messages and/or user prompts may be lost.

## Predefined Numbered Streams

The twenty predefined numbered streams are a set of integers from 0 to 19. A stream designator for a predefined numbered stream may also be given in two ways: one, as an integer expression which evaluates to the appropriate integer value from 0 to 19; or two, as a string expression which evaluates to the appropriate character value (e.g. "5"). The predefined numbered streams are initially attached to the corresponding MTS logical devices 0 to 19, but they may be attached to other files or devices by calling the Assign procedure described below.

## User Defined Streams

The Assign procedure, described later in this section, allows a user program to add a user defined input/output stream dynamically to the set of available streams. The name of a user defined stream may be any non-blank character string containing between 1 and 30 characters. However, the name may not be the same as any of the predefined stream names and may not contain any embedded blanks. A stream designator for a user defined stream may be given as any string expression which evaluates to the name associated with that stream when Assign is called.

When such a dynamically assigned input/output stream is no longer needed, it may be deleted by means of the Release predeclared procedure also described later in this section.

A maximum of 25 user defined input/output stream names may be extant at once.

## Basic Input and Output Streams

The basic input stream is that stream being used by the basic predeclared input procedures Read, Readon and Readcard. The basic output stream is that stream being used by the basic predeclared output procedures Write, Writeon and Writecard, Iocontrol and Newline. By default the basic input and output streams are INPUT and PRINT, respectively. However, the procedures Reader and Writer (described later in this section) may be called to change the basic input/output streams.

There are two predeclared bits variables which may be used used as stream designators and which always refer to the current basic input and output streams. The stream pointer variable Rdr always refers to the current basic input stream, and the stream pointer variable Wtr always refers to the current basic output stream. Note that a program should not assign a value to either Rdr or Wtr. These variables should be used as read-only variables; that is, a program should only use these variables to find out what streams are currently being used by the basic input/output procedures. An example of the use of these variables is given below, near the end of the section "Sense".

## INPUT AND OUTPUT TO A DESIGNATED STREAM

The previous section, "Basic Input and Output," described the use of Read, Write and related basic input and output procedures. These are concerned with input from a single input stream (INPUT) and output to a single output stream (PRINT). An Algol W program may use multiple input/output streams in one of two ways:

(1) By changing the basic input and output streams as needed by calling the predeclared procedures Reader and Writer, as described below.

(2) By using a set of predeclared input and output procedures to specify directly which input or output stream is to be used. These procedures are all described in the section "Stream Directed Input and Output." Some of these stream directed input/output procedures can also be passed a format string to control input/output formatting. This is described in the section "Format Directed Input and Output."

## Changing the Basic Input Stream - Reader

The predeclared procedure Reader may be used to have a designated stream be used as the basic input stream. A call of the predeclared procedure Reader has the general format:

```
Reader(<stream-designator>)
```

When Reader is called, any partially processed input record on the current basic input stream is forgotten. The stream which will be used for input by the basic input/output procedures is changed to the designated one. Any subsequent input using these procedures will be from the new stream, starting with the first character of the next record in sequence.

Example:

```
Reader(3);
for I := 1 until 10 do
begin
   Iocontrol("NEXTCARD");
   for J := 1 until 10 do
      Readon(Data(I, J))
end;
Reader(Input);
```

This example shows the basic input stream being temporarily redirected to the predefined numbered stream 3 while 100 data items are input, 10 to an input record.

## Changing the Basic Output Stream - Writer

The predeclared procedure Writer may be used to have a designated stream be used as the basic output stream. A call of the predeclared procedure Writer has the general format:

```
Writer(<stream-designator>)
```

When Writer is called, any partially built output record on the current basic output stream is written out to that stream. The output stream used by the basic output procedures is then changed to the designated stream. Any subsequent output via these procedures will be to the new stream, starting with the first character of a new record.

If automatic carriage control is on (the default), this first character will be set to a space, giving a new line only. If a different carriage control is required, Iocontrol should be called with a suitable argument before the next Write, Writeon or Writecard call.

Example:

```
Writer(Punch);
Write_Cc := false;
for I := 1 until Numdata do
   Writecard(Cards(I));
Write_Cc := true;
Writer(Print);
```

This example causes the basic output stream to be temporarily redirected to the PUNCH output stream while a set of data records are written there. Note that automatic carriage control character generation is suppressed while the cards are output. This would normally be desired when data is output on cards or to a file.

## DYNAMIC CONTROL OF INPUT/OUTPUT STREAMS

Algol W provides two predeclared procedures, Assign and Release, which may be used within a program to dynamically create and delete user defined input/output streams, and to attach or release files and devices from already existing streams.

### Assign

The predeclared procedure Assign is used either to create a user defined stream and attach an MTS file or device to it, or to attach a file or device to an already existing input/output stream. It may not be used with the named predefined streams. The procedure call takes the form:

```
Assign(<stream-designator>, <file-or-device-name>)
```

Exactly two parameters must always be given. The first parameter specifies the stream to be used and the second the file-or-device-name

to be attached to this stream. The second parameter may be given as any string expression which evaluates to an appropriate file or device name.

If the stream designator does not exist, it is created as a user defined stream. If it does exist, any previous file or device attached to it is released. If it was in use for output, any partially built output record would be written out to it first. If it was in use for input, any partially processed record is forgotten. The new file or device is then assigned to the stream name. The form of the file-or-device name is as required for the operating system concerned. In MTS, any valid MTS file-or-device name is acceptable as the second argument to Assign.

Examples:

```
Assign(3, "W701:DATA");
```

Any assignment to predefined stream 3 will be released and the MTS file W701:DATA assigned to it instead. Stream 3 may be used subsequently for either input or output if the state of the file allows this.

```
Assign("TAPE1", "*T1*");
```

The user defined stream TAPE1 is created and the MTS pseudo-device name *T1* is attached. In this case it would be assumed that *T1* was an MTS $MOUNTed magnetic tape.

## Release

The predeclared procedure Release is used to release a file or device attached to an input/output stream. It may not be used for the named predefined streams. The procedure call takes the form:

```
Release(<stream-designator>)
```

When a predefined numbered stream is specified, it will have no file or device attached to it when Release has finished. When a user defined stream is given, the stream name itself is deleted from the Algol W environment.

If more then one parameter is given, all are presumed to be stream designators and Release will act on each of them in turn.

Examples:

```
Release(9);
```

Any file or device attached to the predefined, numbered stream 9 is released.

```
Release("TAPE1", "TAPE2");
```

The files or devices assigned to these two user defined streams are released and the two stream names are deleted.

## INPUT/OUTPUT STREAM PREDECLARED UTILITY PROCEDURES

Several predeclared procedures are provided to control input/output streams or the files or devices attached to them. These procedures are listed in the following table, and described in more detail below:

| Name | Purpose |
|---|---|
| Rewind | Reset to the beginning of the file or device. |
| Empty | Delete all information from the attached file. |
| Flush | Force out any partially built output record. |
| Protect | Ensure integrity of the attached file. |
| Qualify | Set input/output attributes internal to Algol W. |
| Control | Send device commands to the operating system. |
| Sense | Obtain information about the attached file or device. |

### Rewind

As the name implies, the specified stream will be rewound so that subsequent reads and writes start at the beginning of the file or device attached. Note that, in MTS, only the currently active member of a series of concatenated files or devices will be rewound. An attempt to rewind a stream which may not be rewound (such as a card reader) will cause a fatal error. The named predefined streams may not be rewound.

The procedure call takes the form:

```
Rewind(<stream-designator>)
```

The designated stream is rewound. If more than one parameter is given they are all considered to be stream designators and Rewind will act on each in turn.

Example:

```
Assign(9, "*T*");
Getcard(9, Rec);
 .
 .
Rewind(9);
Getcard(9, Rec);
 .
 .
```

Both Getcard calls (see the section "Stream Directed Input and Output" for the description of Getcard) read in the same record from the predefined input/output stream 9.

Empty

If the specified stream is attached to a file then Algol W will attempt to empty it. Attempts to read an empty file produce an end-of-file indication and the next output procedure call would place a new first record in the file.

The procedure call takes the form:

```
Empty(<stream-designator>)
```

The designated stream is emptied. If more than one parameter is given, they are all considered to be stream designators and Empty acts on each of them in turn. The named predefined streams may not be emptied.

Example:

```
Assign(7, "-QQSV");
Empty(7);
Putcard(7, "Results");
```

The invocation of Empty ensures that the Putcard operation puts the string "Results" into the MTS scratch file -QQSV as the first and only record.

Flush

Flush causes any contents of the output buffer for the designated stream to be written out. It has the same effect on the specified stream as Iocontrol("SPACE") would have on the basic output stream.

The procedure call takes the form:

```
Flush(<stream-designator>)
```

Any partially built output record for the designated stream will be forced out on the file or device attached. If more than one parameter is given each is considered to be a stream designator and Flush will act on each in turn.

Example:

```
Put(Print, "H0,2(3X,I5)", I, J);
Flush(Print);
```

The call to Put builds an output record containing two integers in the internal Algol W storage region associated with the PRINT stream. However the output is said to be buffered, that is the information will not appear on the attached file or device until another output operation on the same stream starts a new output record. Flush causes the record built to be forced out without starting a new output record. This would be done if it was desired to see the result of the Put operation immediately.

Protect

Protect is like Flush in that any partially built output record will be forced out. If the stream is attached to a file, Protect will cause all changes to the file buffers to be written back to the disk copy of the file, so protecting the information from damage caused by an operating system failure occurring at a subsequent time in the program run. Without this protection, the highly unlikely occurrence of a system crash before the file was released could result in information being lost.

The procedure call takes the form:

```
Protect(<stream-designator>)
```

The file buffers for the file attached to the specified stream designator are written to the disk copy of the file to bring it up to date with the program's view of it. If more than one parameter is given, each is considered to be a stream designator and Protect will act on each in turn.

Example:

```
Xdelete("JJJ", 1000);
Protect("JJJ");
```

The first line deletes the internal line number 1000 from the file attached to stream JJJ. The second line ensures that this change is reflected in the disk copy of the file. This reduces the elapsed time

in which an operating system failure could leave the file in a state where internal line number 1000 had not been removed when the program thought it had. It is a request for immediate action where the operating system might otherwise delay the updating for reasons of overall efficiency.

## Qualify

Qualify is related to Control (described later in this section) in that it allows a user to change the attributes of an input/output stream. However, while Control communicates with the operating system, Qualify sets those attributes which are local to Algol W. For instance, it is called to change the lengths of input/output streams.

The procedure call takes the form:

Qualify(<stream-designator>, <qualification-string>)

The qualification string is a command from the program to the Algol W input/output system. It consists of one or more commands, each separated from the next by blanks or a comma. The entire string must be enclosed in quotes ("). Each command is of the form <keyword> or <keyword>=<expression>. Each keyword may be abbreviated down to the minimum unambiguous form or three characters, whichever is the greater. At least two parameters must be given to Qualify. If more than two parameters are passed to Qualify then subsequent ones are assumed to be additional qualification strings and are processed accordingly.

The following keywords may be used in the qualification string.

MAXINPUT=<integer>

Specifies the number of columns of the input record which will be scanned for input data items on this stream.

MAXOUTPUT=<integer>

Specifies the maximum output record length which will be built on the stream by Write, Put, etc. If any attempt is made to build a longer output record then a record of the specified maximum length will be forced out and a new one started.

MCC

This tells Algol W that output records are being prepared with IBM machine carriage control. It is the programmer's responsibility to provide the relevant carriage control characters as the first character of each output record.

-MCC

See ASA.

ASA

Tells Algol W that output records are being built with the American Standards Association carriage control. This is the converse of MCC and is the default.

IC

IC stands for implicit concatenation which means allow implicit transfer from one file or device to another by a directive within the file-or-device records (such as the MTS $CONTINUE WITH directive). This the default.

-IC

Prevents implicit concatenation - see IC above.

PAGELIMIT={ON|OFF} or PL={ON|OFF}

This parameter only applies if EPAGES=<integer> has been given as a compilation or runtime parameter. PAGELIMIT alone or PAGELIMIT=ON enables page limit checking for the designated stream. PAGELIMIT=OFF disables this check. If a page limit has been given then PAGELIMIT defaults to ON for the PRINT stream and OFF for all other streams.

CTRETURNS

This keyword specifies that system errors encountered during execution of the Control predeclared procedure (see below) should not cause a fatal run error. Such conditions place the system Control return code in the predeclared integer Syscode and execution of the calling program continues.

CTFIELDED

Undoes the effect of the previous keyword, CTRETURNS. System errors encountered during execution of the Control predeclared procedure cause a fatal run error. This is the initial state.

Examples:

```
Qualify(Input, "MAXI=80");
for I := 1 until 100 do
   Readon(Data(I));
```

Here the Qualify call specifies that only columns 1 to 80 of the records read in on stream INPUT are to be scanned. The effect is as if columns

81 to the end of the input records are always blank. The 100 data items read in will be from the first 80 columns of the data records only.

```
Qualify(Print, "PL=OFF");
Qualify(9, "PL=ON");
```

If a page limit has been set through a compilation or runtime parameter then these two Qualify procedure calls transfer the check away from the default PRINT stream to predefined stream 9.

## Control

Control allows a program to send device control commands to the operating system to cause the state of a file or device attached to an input or output stream to be changed. The format of the commands is described in other documents (e.g. MTS Volume 4, "Terminals and Tapes").

The form of the procedure call is :

```
Control(<stream-designator>, <control-string>)
```

<control-string> is passed on to the operating system device control routines which act on the file or device attached to the designated stream. At least two parameters must be given. If more than two are given then the subsequent ones are assumed to be control strings and each is passed to the operating system in turn. Each control string is translated to upper case before transmission to the operating system device support.

Examples:

```
Assign(9, "*TAPE*");
Control(9, "POSN=*15*");
```

In this example *TAPE* is assumed to be an MTS $MOUNTed magnetic tape. The Control procedure causes the string POSN=*15* to be sent to the magnetic tape device support routines in MTS. This causes the tape to be positioned at the start of the 15th file (see MTS Volume 4: "Terminals and Tapes" for further details).

```
Putcard(Error, "0Enter password");
Control(User, "BLANK");
Getcard(User, Password);
```

This sequence is used to read in a password to a program in MTS without echoing the password as it is typed in. To achieve this, the control procedure sends the terminal device command %BLANK to the stream from which the read is attempted before requesting the read. Note that the device command character, %, is not included in the argument to Control.

## Sense

Sense is used to return information about the state of an input/output stream and the file or device attached to it. Essentially it is an input operation similar to Get (described in the section "Stream Directed Input and Output") but the input is information about the file or device rather than records supplied by it.

The procedure call takes the form:

Sense(<stream-designator>, <sense-request-string>, <sense-list>)

The <sense-request-string> is a list of keywords requesting particular items of information about the file or device attached to the designated stream. The list may contain one or more keywords, each separated from the next by a comma. The entire list must be enclosed in quotes ("). Each keyword may be abbreviated down to the minimum unambiguous form or three characters, whichever is the greater. The <sense-list> consists of one or more variables, each separated from the next by a comma. For each keyword in the <sense-request-string> there must be a receiving variable in the <sense-list>. There must be at least one keyword in the sense request string and Sense itself must have at least three parameters.

The keywords of the sense request string are as follows:

STREAM

This returns the Algol W stream name as specified by the stream designator. The only valid data type for the receiving variable is string(30).

LSTREAM

This returns the length of the stream name returned when the STREAM keyword is given. The only valid data type for the receiving variable is integer.

FDNAME

This returns the name of the file or device attached to the specified stream designator. The only valid data type for the receiving variable is string(44).

LFDNAME

This returns the length of the file or device name returned when the FDNAME keyword is given. The only valid data type for the receiving variable is integer.

TYPECODE

This returns a four character device type describing the file or device attached to the designated stream. If nothing is attached the type is NONE. For an MTS line file the type is FILE; other MTS types will be found in the GDINFO subroutine description in MTS Volume 3, System Subroutine Descriptions. The only valid data type for the receiving variable is string(4).

For the next six keywords a logical value is returned into a logical, integer or bits variable. True is signified by 'true', 1 or #1. False is signified by 'false', 0 or #0. If the specified stream is not assigned all six keywords return 'false', 0 or #0.

INPUT

This returns 'true' if the attached file or device may be used for an input operation.

OUTPUT

This returns 'true' if the attached file or device may be used for an output operation.

REWIND

This returns 'true' if the attached file or device may be rewound.

INDEX

This returns 'true' if the attached file may be used in an indexed input/output operation. Indexed input/output may be performed by the predeclared procedures Xgetcard, Xputcard and Xdelete.

DEFAULT

This keyword only applies to predefined streams. It returns 'true' if, in the $RUN command which invoked the program, the system logical device name equivalent to the designated stream was left to default, that is it was not explicitly assigned.

CONCATENATION

This returns 'true' if the file or device attached to the designated stream is a member of an explicit concatenation of file-or-device names and the currently active member of the concatenation is not the last in the sequence.

The following six keywords return length information. Only receiving variables of type integer are valid. If the specified stream is not assigned all six keywords return zero.

POSINPUT

Returns the value of the next input pointer in the input buffer. This is the integer displacement from the start of the buffer from which the next input operation will start fetching characters.

POSOUTPUT

Returns the value of the next output pointer in the output buffer. This is the integer displacement from the start of the buffer at which the next output operation will start placing characters.

MAXINPUT

Returns the length of the input buffer which will be scanned for input data. This is the value which may be set by the MAXINPUT keyword with the Qualify predeclared procedure.

MAXOUTPUT

Returns the length of the output buffer which will be used when building output records. Attempts to output longer records will result in the text overflowing to another output record (or records). This is the value which may be set by the MAXOUTPUT keyword with the Qualify predeclared procedure.

SYSINPUT

Returns the maximum length of any input record which may be expected from the attached file-or-device. An empty file will return zero; a file containing information will return the length of the longest record to exist since it was last emptied; devices return their physical maximum input length.

SYSOUTPUT

Returns the maximum length of any output record which may be written to the attached file-or-device. Any record longer than this will be truncated at the right. This keyword can be used to determine terminal output lengths for purposes of formatting output.

The remaining two keywords return system pointers.

SYSBLOCK

This returns an entity which the operating system uses as a reference to the attached file or device name when performing input/output operations. In the case of MTS this is the FDUB (file-or-device usage block) pointer. The valid data types for the receiving variable are integer or bits.

IOCBLOCK

This returns the address of the Algol W system internal control block used for input/output in connection with the designated stream. This is only of interest to persons maintaining the Algol W system but is documented here for completeness. The valid data types for the receiving variable are integer or bits.

Examples:

```
Sense(Wtr, "STREAM", Strsave);
Writer(Punch);
for I := 1 until Cardsout do
   Writecard(Cards(I));
Writer(Strsave);
```

In this example Sense is used to find the stream name of the basic output stream. The PUNCH stream is then made the basic output stream and Cardsout records are output. A second call to Writer restores the returned original stream name as the basic output stream.

```
Sense(9, "TYPECODE, FDNAME, LFDNAME", Tcode, Fdn, Lenfdn);
if Tcode="NONE" then
   Write("Stream 9 is not assigned")
else
   Write("File = ", Fdn, Lenfdn, "Characters");
```

This example prints the state of stream 9.

## Sense, FDUB-Pointers, and MTS File Locking

In MTS, explicit file locking is a commonly used method of implementing processes which share and modify the same file by two or more tasks in turn. The MTS subroutines LOCK and UNLCK provide the necessary control over file access to achieve the required Dijkstra semaphore operations - if used carefully.

Both of these subroutines require the FDUB-pointer for the file to be locked. This can be obtained by a Sense statement which requests the SYSBLOCK information, as described in the previous section. However it is important to realize that Sense opens the file and therefore leaves it locked for reading. The sequence:

```
Assign("Q", "W701:SHARE");
Sense("Q", "SYSBLOCK", Fdub);
Call("UNLCK", Fdub);
```

is recommended when it is desired to open a file and obtain the FDUB pointer without leaving the file locked afterwards. See the description of LOCK and UNLCK in MTS Volume 3, System Subroutine Descriptions for

further details of these subroutines. The Call predeclared procedure is described in the section "External Linkages."

The technique used when co-operating processes compete for the same file is as follows:

(1) A process requiring a file must first request it. This takes the form of a LOCK subroutine call. If the operation must only read from the file, lock the file for reading. Any kind of modification must request locking for modification. The call of LOCK normally would request an indefinite wait for the file.

(2) Check the return code from LOCK. If it is non-zero do not proceed further. Return codes are also described in the section "External Linkages."

(3) Any input/output operation on the file is then performed. LOCK will not return control with a return code of zero unless this is possible. It is essential that no action is attempted which may have to wait on any other process while the file is locked. This includes any kind of input/output, using other files or a conversational terminal.

(4) Call UNLCK to release the file for other processes.

If the Release predeclared procedure is used to free an Algol W stream, the FDUB pointer is released as well. This means that Release will unlock the file, so in this case there is no need to call the MTS UNLCK entry.

# STREAM DIRECTED INPUT AND OUTPUT

The section "Basic Input and Output" discussed procedures which perform input/output via the basic input and output streams. The definition of these streams and of the input/output stream designators in general was given in the section "Multiple Input and Output Streams."

This section describes a set of predeclared input/output procedures with which the stream to be used in an input/output operation may be specified directly. These provide several additional facilities beyond those given by the simple input/output routines:

(1) A more convenient mechanism is provided for detecting end-of-file.

(2) Facilities are provided for indexed input/output operations. A program may specify a record index or line number to be used in the input/output operation, where the stream concerned has a file attached which supports direct access operations (for example MTS line files). Records may also be deleted from a file.

(3) For those routines which edit individual items in building or decoding input/output records, the operation may be performed either as a free format one (like Read or Write) or under the control of a specific format string. These format strings are similar to, but by no means identical with, facilities provided by the Fortran language.

(4) The output routines described in this section never generate carriage control characters automatically; these must be specified explicitly if required.

## INPUT AND OUTPUT OF COMPLETE RECORDS

Two predeclared procedures are provided, analogous to Readcard and Writecard, which allow the input or output of complete records from a specified input/output stream.

Getcard

Getcard fetches one or more records from the specified input stream. Any partially processed input record for the stream (such as those fetched by Read, Readon, Get or Geton) is discarded. The next record is input into a supplied string variable in its entirety without any editing. After the operation is complete any subsequent input operation of any kind from that stream (including Readon or Geton) will cause a new input record to be fetched.

The procedure call takes the form:

```
Getcard(<stream-designator>, <input-string-variable>)
```

At least two parameters must be given. After the operation the <input-string-variable> will contain the contents of the record fetched. If the input string variable has been declared with a length greater than that of the record fetched, the characters will be padded on the right with blanks. If, on the other hand, it has been declared with a length less than that of the input record, then the input record is truncated at its right hand end. If more than two parameters are given, the third and subsequent ones are assumed to be further input string variables and another input record is fetched for each one.

End-of file handling for Getcard never uses the predeclared reference variable Endfile and its related exception processing mechanisms. Instead, any input operation performed by Getcard causes the predeclared logical variable Filemark to be set. This will be 'true' if the end-of-file has been detected, in which case the input string variable is returned completely filled with blanks.

Example:

```
string(256) Recd;
while
begin
   Getcard(User, Recd);
   ¬Filemark
end do
begin
    .
    .
end;
```

In this example input records are fetched, without editing, from the USER input stream. Here, Getcard as issued is part of a <block-expression> with an end-of-file indication as the value of the expression. Input records are therefore fetched until end-of-file is signalled for the input stream. For each record fetched successfully the block of statements forming the 'do' clause will be executed. This example is typical of the type of coding which might be used in a program designed to process commands entered by a user.

Putcard

Putcard sends one or more complete records to the specified output stream. Before the operation any partially built output record for the stream (such as those constructed by Write, Writeon, Put, Puton or Newline) is forced out. Then the supplied record is output from a string expression. It is written immediately to the specified output stream as if the Flush predeclared procedure had been called. This means that any subsequent output operation on this stream (even by Writeon or Puton) will start a new output record.

The procedure call takes the form:

```
Putcard(<stream-designator>, <output-string-expression>)
```

At least two parameters must be given. The contents of <output-string-expression> are written to the designated stream. If more than two parameters are given then the third and subsequent ones are also assumed to be output string expressions and a separate output record is written for each.

Note that Putcard will never generate carriage control characters automatically. If these are desired they must be explicitly specified as the first character of the output string expression.

Example:

```
for I := 1 until Numdata do
   Putcard(Punch, Cards(I));
```

This example shows Putcard being used to output a set of strings to the PUNCH output stream. It is similar to the example given for the use of the Writer procedure (see "Changing the Current Output Stream - Writer" in the section "Multiple Input and Output Streams") but is more concise to program.

INDEXED INPUT AND OUTPUT

Algol W provides the means to do indexed input/output operations on streams attached to files which support direct access by record index or line number. The routines are similar to those described in the previous section (Getcard and Putcard) but in each case an additional parameter specifies the record index or line number to be used in the operation. Note that Algol W can only perform indexed input/output on complete records.

If input editing of the kind provided by Read, Readon, Get or Geton is required during an indexed operation then the complete record should be fetched and decoded using the string conversion predeclared procedure Getstring.

If output formatting of the kind provided by Write, Writeon, Put or Puton is desired during an indexed operation then the record should be formatted into a string using the predeclared procedure Putstring and then the complete string output in an indexed operation.

Getstring and Putstring are described later in this section.

The index parameter is always an integer. In the case of MTS line files it specifies the internal line number, which is an integer one thousand times the external line number. The external line number is that printed by such commands as $LIST or $EDIT, in which up to three places of decimals may appear.

## Xgetcard

Xgetcard is used to read an input record from a specified record index or line number.

The procedure call takes the general form:

Xgetcard(<stream-designator>,<record-index>,<input-string-variable>)

At least three parameters must be given. An input record is fetched into the <input-string-variable> from the specified stream as if it was a Getcard request, but the record is fetched from the position in the file given by the <record-index>. The effect of using Xgetcard is to reposition the next-input record pointer for the file so that any subsequent sequential input operations continue from the point in the file just after the record fetched by Xgetcard.

If more than three parameters are given, then the fourth and subsequent ones are assumed to specify further input string variables. However only the input string variable specified by the third parameter is used as the subject of an indexed input operation. Input records are read in sequence into the input string variables specified by the subsequent parameters as if they were the subject of a Getcard call.

Xgetcard cannot suffer from an end-of-file condition as such. However it is possible that no record may be present in the file at the position indicated by the record index. Should this happen, it is treated as a pseudo end-of-file condition and treated in the same way as Getcard would treat a true end-of-file. That is, the predeclared variable Filemark has the value 'true' when Xgetcard returns and the input string is filled with blanks.

Example:

```
Index := 1;
while begin
   Xgetcard(9, Index*1000, Recd);
   ¬Filemark
end do
begin
   Write(Index, Recd);
   Index := Index + 1
end;
```

In this example Xgetcard is being used to read in lines 1.000, 2.000, 3.000 and so on from an MTS line file until a line with the required number cannot be found. Any records present at any of the 999 file lines possible between each integer line number are ignored. For each line found the 'do' clause writes out the line number and the contents of the record. A possible application of this seemingly trivial example would be in a program which maintained a catalog of information. This might well be kept in a file structured so that the heading for each entry was held at an integer internal line number, while the information for each entry was stored on the subsequent decimal line positions. The effect of this section of the program would then be to print a contents list for the catalog without printing the full text for each entry.

## Xputcard

Xputcard is used to write a record to a specified record index or line number in the output file.

The procedure call takes the form:

Xputcard(<stream-designator>,<record-index>,<output-string-expression>)

At least three parameters must be given. A record is written from the <output-string-expression> to the designated stream as if it were a Putcard request, but the record is written to a position in the file given by the <record-index>. The effect of using Xputcard is to reposition the next-output record pointer for the file so that any further sequential output operations continue from the point in the file just after the record written by Xputcard.

If more than three parameters are given then the fourth and subsequent ones are assumed to specify further output string expressions. However only the output string specified by the third parameter is used as the subject of an indexed operation. Records are written in sequence from the output string expressions specified by the subsequent parameters as if they were the subject of a Putcard call.

Note that Xputcard, in common with the other output routines described in this section, never supplies a carriage control character. If this is desired it must be supplied by the program as the first character of the output string. However this should be rare with Xputcard since its normal purpose would be to build data structures in files rather than to prepare text for direct printing.

Example:

```
Xputcard(9, Index*1000, Info_Label);
for I := 1 until Info_Lines do
   Xputcard(9, Index*1000 + 1, Info(I));
```

Continuing the catalog example from the previous section, this code could be used to insert a single entry. The first use of Xputcard writes the heading line to an integer external line number specified by Index. Then the Xputcard which is the subject of the For statement follows this with a series of lines at the internal line number interval one (external 0.001). These form the text of the entry.

## Xdelete

Xdelete is used to delete lines from a file when they are no longer needed. The requirement is to write a line of length zero to the file. Xputcard cannot do this because a string must be of length at least one, so this predeclared procedure is supplied to perform the required action.

The procedure call takes the form:

```
Xdelete(<stream-designator>, <record-index>)
```

At least two parameters must be supplied. The record specified by <record-index> is deleted from the file. This call is valid whether or not a record of this line number exists in the file: it is not an error to delete a non-existent record. If more than two parameters are given, the third and subsequent ones are also each assumed to be a record index and the relevant lines are deleted from the file in turn. Note that all of these parameters perform an indexed operation. Since Xdelete performs an output operation like Xputcard, the next-output record pointer is set to point at the record following the last record deleted.

Example:

```
Read(Index);
Xdelete(9, Index*1000);
```

Again continuing the example of a catalog, this example might read in a catalog index number for deletion. The call to Xdelete deletes the record which is the header for the catalog item starting at the integer

external line number. Note again the multiplication factor of one thousand to provide the required internal line number as the parameter.

## Returned Line Numbers

It is often useful to be able to read sequentially through a file while obtaining the line number at which each record is stored as it is read. Algol W provides a means for doing this.

For any input/output operation performed by Algol W, a predeclared integer variable, Sysindex, is set to the internal line number used in the operation. This is most useful when used in connection with calls to Readcard, Writecard, Getcard and Putcard. When used in connection with buffered input/output, it is essential to remember that Sysindex is set as each record is fetched or flushed and not as it is decoded or built. Remember also that the value of Sysindex must be saved or used before any other input/output operation is performed, as this would, of course, reset it.

Example:

```
begin
   string(256) Recd;
   integer Saveindex;
   while
   begin
      Getcard(0, Recd);
      Saveindex := Sysindex;
      ¬Filemark
   end do
      Xputcard(1, Saveindex, Recd);
end.
```

The Getcard within the block expression reads in records from stream 0 until the end-of-file is detected. The Algol W system will set the predeclared variable Sysindex each time a record is read. This is used in the 'do' clause to supply the record index for the Xputcard writing the record to stream one. The effect is to copy the file on stream 0 to the file on stream 1 preserving the original line numbers. If stream 0 was attached to a file called DATA and stream 1 was attached to a file called BACKUP, the effect would be as if the MTS command:

```
$COPY DATA BACKUP@INDEXED
```

had been entered by the user.

## INPUT AND OUTPUT OF INDIVIDUAL ITEMS

Algol W provides four predeclared procedures, Get, Geton, Put and Puton, which allow the input/output of individual items to a specified stream. Get and Geton are analogous to the simple input procedures Read and Readon. Put and Puton are analogous to the simple output procedures Write and Writeon. These routines may either perform free format input/output (as do the simple input/output procedures) or their action may be controlled by a supplied format string. Format string processing is described in the section "Format Directed Input and Output."

### Get and Geton

The general form of these procedure calls is:

```
Get(<stream-designator>, <format> [,<get-list>])
Geton(<stream-designator>, <format> [,<get-list>])
```

where:

square brackets, [ ], specify an optional sequence;

<format> is either the reference constant 'null' or a string expression known as a format string; and

<get-list> is a list of one or more variables, each separated from the next by a comma and known as a <get-item>.

Get will always fetch a new input record from the specified stream. Geton will continue from where the last input operation finished.

If the format is given as the reference constant 'null' then the input operation is a free format one and items in the <get-list> are input as if they were parameters to Read and Readon. If the contents of the input record being processed are exhausted then the next record in sequence will be read in and decoding continues from the first byte of the new record.

If the format parameters are given as a string expression, this is interpreted as a format string and is used to control the input of data to items in the <get-list> as described in the section "Format Directed Input and Output."

If input is proceeding under the control of a format string and the information in the string is exhausted but there are still items in the <get-list> requiring data, then the remainder of the operation is completed in free format as if 'null' had been specified, that is as for Read or Readon but from the specified stream.

At least two parameters must be given. Note that the <get-list> is optional because a supplied format string may only cause input records to be skipped or the pointer to an input record to be moved. If this were true no actual input conversion would be performed.

End-of-file is handled in the same way as for Read and Readon, by inspection of the predeclared reference Endfile.

Note however that, if an end-of-file occurs as the result of the initial read of a Get or as the result of format string action, then the end-of-file condition is not processed immediately. Processing of the condition is deferred until an actual data item is requested (and cannot be fetched), even if this means waiting for a new input procedure call for the stream. If a request for indexed input (via Xgetcard) is issued while an end-of-file is pending for the stream, the pending condition is reset.

Examples:

```
integer A, B, C;
Get(3, null, A, B, C);
```

This example reads three integers into the variables A, B and C from predefined stream 3. The operation is a free format one. Records would be input and scanned as necessary in order to fetch the next three data items in the same manner as the predeclared procedure Read.

```
real X, Y;
Get(Input, "2(5X,E18.0)", X, Y);
```

This example would read in a record from the input stream and decode from it two real numbers into the variables X and Y. This decoding is done under the control of the format string supplied as the second parameter. The format codes used are described in the section "Format Directed Input and Output."

## Put and Puton

The general form of these procedure calls is:

```
Put(<stream-designator>, <format> [,<put-list>])
Puton(<stream-designator>, <format> [,<put-list>])
```

where:

square brackets, [ ], specify an optional sequence;

<format> is either the reference constant 'null' or a string expression known as a format string; and

<put-list> is a list of one or more expressions each separated from the next by a comma and known as a <put-item>.

Put will always start a new output record for the specified stream. Puton will continue from where the last operation on this stream finished.

If the format is given as the constant reference 'null' then the output operation is a free format one and items in the <put-list> are output as if they were parameters to Write and Writeon. If there is no more room in the current output record for a <put-item>, the record is output and a new one is started.

If the format parameter is given as a string expression then this is interpreted as a format string and output of data from items in the <put-list> proceeds under its control as described in the section "Format Directed Input and Output."

If output is proceeding under the control of a format string and the information is exhausted but there are still items in the <put-list> to be output, then the remainder of the operation is completed in free format as if 'null' had been specified, that is as for Write and Writeon but with the specified stream.

At least two parameters must be given. Note that the <put-list> is optional because a supplied format string may only cause output records to be skipped or add spaces or literal character strings to the output record for a format string.

These procedures never supply a carriage control character automatically. If one is desired then the program must insert it as the first character of the record.

Examples:

```
Put(Print, null, "0The value of pi is ", Pi);
```

This example writes out to the PRINT stream the given string followed by the long real predeclared variable Pi in the same way as Write would have done. For instance the output conversion of the value of Pi is done under the control of the predeclared format variables R_Format, R_W, R_D, R_Expchar, S_W or R_Sig as appropriate. Note that the supplied string expression starts with the character 0. This is an explicitly supplied carriage control character to skip two lines before writing the subsequent characters.

```
for I := 1 until 10 do
begin
   Put(7, "5X");
   for J := 1 until 10 do
      Puton(7, "F8.3,X", Data(I,J))
end;
```

This example shows the use of Put and Puton to dump 100 real values for an array Data on the predefined stream 7. Full details of the formats used will be found in the section "Format Directed Input and Output." Notice that the initial Put has no items in its <put-list>. It simply starts a new output record with 5 spaces.

## INTERNAL INPUT AND OUTPUT CONVERSION

Algol W provides aids to the programmer who wishes to perform formatted input/output conversion to or from string variables declared within a program. One possible use of this facility would be to build or decode strings used as input/output records by the indexing routines Xgetcard and Xputcard. Another possible application would be to build complex system commands for use by the Control or Cmd predeclared procedures.

This section discusses the routines Getstring and Putstring. However in this context it should also be noted that there are a set of primitive predeclared string conversion functions:

```
Code
Decode
Intbase10
Intbase16
Base10
Longbase10
Base16
Longbase16
```

These routines are described in the section "Strings."

### Getstring

Getstring is analogous to Get but the source of the input data is an Algol W string expression rather than an input stream.

The general form of the procedure call is:

```
Getstring(<input-string>, <format>, <get-list>)
```

where:

<format> and <get-list> are as described earlier in this section under "Get and Geton"; and

<input-string> is a string expression.

There would be little point in specifying a constant input string since the purpose of the routine is to decode the contents of a string into the individual items in the <get-list>. At least three parameters must be given. For this procedure there would be no point in not having a <get-list> since no input conversion would be done.

Getstring normally starts decoding from the first character of the input string, with no pointer maintained for the input string once the Getstring call is completed. This implies that all of the items required for a particular input string must be fetched in a single call to Getstring. The action of the procedure in this respect can however be modified; see the section "Control of Getstring Action" in the section "Miscellaneous Topics."

If the format is given as the constant reference 'null' then the input operation is a free format one. Scanning of the input string proceeds as if this was a call to Read. However since the source of the data is a single string rather than a stream, no new record can be fetched if the string is exhausted. Should this condition occur, it is normally treated as a fatal error condition.

If the format parameter is given as a string expression then this is interpreted as a format string and input of data to items in the <get-list> proceeds under its control as described in the section "Format Directed Input and Output."

Any slashes (/) in the format string are ignored.

If input is proceeding under the control of a format string and the format information is exhausted but there are still items in the <get-list> requiring input data, then the remainder of the operation is completed in free format as if 'null' had been specified. If the format string specifies a position beyond the end of the input string this is also normally treated as a fatal error.

Example:

```
Xgetcard(9, 1000, Data);
Getstring(Data, "5X,A15,2X,I4", Name, Num);
```

In this example Xgetcard is being used to read in internal line number 1000 from predefined stream 9. Since this is an indexed operation formatted input conversion cannot be done during the actual input operation; only complete records can be fetched. Instead Getstring is called to input a string and an integer into the variables Name and Num using as its source the string fetched by Xgetcard. See the section "Format Directed Input and Output" for an explanation of the format string.

Putstring

Putstring is analogous to Put but the destination of the output data is an Algol W string variable rather than an output stream.

The general form of the procedure call is:

Putstring(<output-string>, <format>, <put-list>)

where:

<format> and <put-list> are as described earlier in this section under "Put and Puton"; and

<output-string> is a string variable.

At least three parameters must be given. For this routine there would be little point in having a null <put-list> since no output conversion would be done.

Before the operation the entire string variable is cleared to blanks. This has the effect of padding any formatted output on the right with blanks.

Putstring always starts outputting the encoded data as the first character of the string variable. No pointer is maintained for the output string once the Putstring call is completed. This means that all of the items which are to be output to a particular string must be included in a single call to Putstring.

If the format is given as the reference constant 'null' then the output operation is a free format one. Encoding of the output string proceeds as if this was a call to Write. However since the destination of the data is a single string rather than a stream, no new string can be started if the current one is full. Should this condition occur the current item will be truncated on the right. Subsequent items in the <put-list> will be ignored.

If the format parameter is given as a string expression then this is interpreted as a format string and output of data from items in the <put-list> proceeds under its control as described in the section "Format Directed Input and Output."

Any slashes (/) in the format string will be ignored.

If the output is proceeding under the control of a format string and the information in the format string is exhausted, but there are still items in the <put-list> to be output, then the remainder of the operation is completed in free format as if 'null' had been specified.

Example:

```
Putstring(Devcom, "'POSN=*',J,H*", Filenumber);
Control("TAPE", Devcom);
```

In this example the Putstring operation builds an MTS magnetic tape device control command in the string Devcom. This is then used as the subject of a Control command to position the tape at the beginning of the file whose number is in the variable Filenumber. If Filenumber contains 23 then the Control statement command would be equivalent to:

```
Control("TAPE", "POSN=*23*");
```

An explanation of the format string used may be found in the section "Format Directed Input and Output."

# FORMAT DIRECTED INPUT AND OUTPUT

## INTRODUCTION TO FORMAT STRINGS

The section "Stream Directed Input and Output" described a set of predeclared input/output procedures whose second parameter designates a format which controls the action of the procedure. They are:

```
Get
Geton
Getstring
Put
Puton
Putstring
```

So far all descriptions of these procedures have used the reference constant 'null' as the format parameter. This implies that the operations are to be done in free format, that is the same editing algorithms used by Read and Write are to be employed. If this second parameter is specified as a string expression then Algol W will use the resulting string as a format directive which is to control the operation. Format strings are constructed in a manner similar to those in Fortran. Note however that Algol W input will never "assume" a decimal point or interpret a blank as a zero.

The following small program gives a simple example of the use of a format string for output. Full details of all format string action are given later in this section.

```
begin
   integer I, J;
   real A;
   I := 34;  J =: -56;  A := 56.7895;
   Put(Print, "X,2(I3,2X),F5.2", I, J, A);
end.
```

The output from this program is a single line written to the PRINT stream, MTS SPRINT, as follows:

```
␣␣34␣␣-56␣␣56.79
```

The format string contains three main items:

```
X
2(I3,2X)
F5.2
```

The commas act as separators. X is the format code for a space and is responsible for the first space in the output line, which prevents an accidental line or page skip. This is particularly important when using procedures such as Put and Puton because they do not generate a carriage control character automatically.

The third format item, F5.2, specifies that a single floating point value is to be output in a field whose total width is 5 character positions with 2 places of decimals after the decimal point, hence "56.79".

The second format item is a little more complex. In 2(I3,2X), the initial 2 is called a replication factor and indicates that the format group within the parentheses is to be obeyed twice before proceeding to the next item. Within the parentheses there are two items:

```
I3
2X
```

I3 specifies that an integer value is to be output in a field 3 character positions wide. 2X specifies that 2 spaces are to be output. Since items are taken from the <put-list> in order, the values of I and J are used to output two fields "Ø34ØØ" and "-56ØØ".

## FORMAT STRINGS

Format strings are composed of format items, each of which may take one of two forms:

(1) a group of characters one of which is a single alphabetic character format code,

(2) a group of characters within primes which form a literal character string.

Format items are separated from one another by a delimiter which is either

(1) a comma, which acts only as a separator;

(2) a slash (/), which separates two format items and also terminates processing of the current input or output record, that is it causes a new line on output and a new physical record to be fetched on input.

## Format Codes

The following is a list of the valid format codes. In their description r indicates that a replication factor may be used with the code, w is an integer number describing the field width, d is an integer number describing the number of decimal places and c may be any single character. The format code may be entered in upper or lower case.

| | |
|---|---|
| rAw | String input/output. |
| rBw | Binary input/output. |
| rDw \| rEw | Floating point input/output. On output, an explicit exponent is printed. |
| rFw.d | Floating point input/output. On output no exponent is printed. |
| rHc | A single character c following the format code is output. On input it behaves as format code X. |
| rIw | Integer input/output. On output the number is right justified in the field. |
| rJw | Integer input/output. On output the number is left justified in the field. |
| rLw | Logical input/output. |
| Tw | Tab format, w specifies the new column position in the input or output record. |
| wX \| Xw | w character positions are skipped in the input or output record. |
| rZw.d | Hexadecimal input/output. |

Literal strings may also be preceded by a replication factor.

## Constructing Format Strings

Format strings are constructed from the individual format items described above.

When a group of format items is to be repeated, the group should be enclosed in parentheses, immediately preceded by the appropriate replication factor. Parentheses may be nested to a depth of 8.

No spaces may appear within a format item but as many as desired may be placed between an item and the leading or trailing comma or slash delimiter, and between the parentheses and the items they enclose. Such spaces can improve readability.

A special data driven replication factor, R, may be used to prefix H, T and X format codes and literal strings for output operations only. In this case, the field width or tab position is supplied as an integer expression which is the next item in the <put-list>. Use of this special factor is described later in this section.

A literal string as a format item consists of any sequence of characters delimited by primes (’). If a prime is desired as a character within a literal string then two primes must be supplied.

## Interpretation of Format Strings

Format strings are interpreted from left to right. Rescanning of any item takes place only under the control of a replication factor. For each format item which acts on data supplied in the <get-list> or <put-list>, the format code must be compatible with the simple type of the relevant variable or expression. Otherwise a fatal error condition is recognized, as also happens when the format code is not a recognized type or if any of the rules for constructing a format string are violated.

When a procedure call is processed, its format string is checked for correctness and interpreted until a <get-item> or <put-item> is required. The <get-list> or <put-list> is then processed. For each item, the pending format item controls the input or output editing performed. If successful, processing of the format string continues until another item from the list is required.

If the <get-list> or <put-list> is exhausted before the format string has been completely processed, then processing of the predeclared procedure terminates and program control passes to the next statement in line. If, on the other hand, the format string is exhausted but there are still items to process in the <get-list> or the <put-list>, execution continues in free format, that is processing continues as if the format had been ’null’.

## FORMAT DIRECTED INPUT

The following sections describe the action of each format in input operations.

"/" Format

A slash symbol encountered in the format string causes the processing of the current input record to be terminated and a new physical input record to be fetched.

For example:

```
Get(9, "I5///A20", I, Str);
```

The three slash symbols cause three new physical input records to be fetched between the input conversions for I and Str.

Literal String Format

This format is provided for use mainly with output procedures. If encountered during format directed input, the number of characters specified in the string is calculated and the internal pointer to the physical input record is moved forward this number of positions. This effect may be achieved more simply using the "X" format code.

"A" Format

"A" format is used to input characters into the next item in the <get-list>. The only valid simple type for the <get-item> is string.

The number of characters fetched from the input record is determined by the field width specified after the format code. If this is zero or omitted, the implied length of the <get-item> is used instead. Where the field width is specified its value should be between 1 and 256. If more characters are fetched than will fit into the receiving string then the characters are truncated on the right. Conversely, if fewer characters are fetched, these are padded on the right with blanks. If when characters are being fetched from the input record, the current record is exhausted, then subsequent records will be fetched until sufficient characters have been obtained.

For example:

```
string(24) S, T;
 .
 .
Get(3, "A8,A", S, T);
```

In the above example eight characters are fetched from a new input record on stream 3 and placed in the string S. The next 24 characters

are placed in the string T since no field width is specified with the second "A" format. If the input record contains:

```
0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

then the values of the variables will be:

```
S = "01234567␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣"
T = "89ABCDEFGHIJKLMNOPQRSTUV"
```

Note the trailing spaces appended to the data value of the first string S.

## "B" Format

"B" format will input data bytes, without editing, into the next data item in the <get-list>. Simple types other than reference are valid.

For the input of strings the behavior of "B" format is almost identical to that of "A" format; the only difference being that, if any padding is necessary on the right, the pad characters will be hexadecimal 00 rather than spaces.

For other data types the number of characters to be fetched is determined by the internal length implied by the <get-item>. These lengths are:

| | |
|---|---|
| integer | 4 |
| real | 4 |
| long real | 8 |
| complex | 8 |
| long complex | 16 |
| logical | 1 |
| bits | 4 |

For these simple types any width specified with the format is ignored: the lengths given in the above table are used instead. If the current physical input record contains insufficient bytes to satisfy the next input operation, it is discarded and a new record is fetched. The entire input field is then taken from this record, starting in column 1. Only string fields may be broken across several input records.

"B" format is designed to allow internal data to be output without editing so that it may be re-input subsequently without any loss of accuracy due to rounding or other causes. Note that the internal forms of numerical and logical values so output may not be readily interpreted unless inspected in a hexadecimal dump. Number representation is discussed in Appendix J.

All values input into real, long real, complex or long complex variables are normalized during the input conversion operations.

## "D", "E", and "F" Formats

These three formats are used to input floating point values into the next item in the <get-list>. While they perform different functions when used as output formats, all three cause the same effect during an input operation. Simple types real, long real, complex and long complex are valid.

For real and long real types, the field width specified, or otherwise one byte, will be fetched from the physical input record. If the current record does not have a field of this width remaining, it is discarded and a new record is fetched. The field is then read from this record starting in column 1. Algol W will attempt to decode a single floating point value from within this field in the same form as that expected in free format input. An error condition is recognized if more than one data item is found within this field.

Example:

```
real A, B;
Get(3, "2E8", A, B);
```

If the input field is

```
ø5.6E8øø6.7øø8.99
```

then A will receive the value 5.6'8 since this is alone in the specified 8 byte field "ø5.6E8øø". When Algol W attempts to decode a value for B, the 8 byte field fetched will contain "6.7øø8.9". Since the field contains two data items a run error condition will be recognized. Had the input record been:

```
ø5.6E8øø6.7øøøøø8.99
```

then B would have received the value 6.7 and, at the end of the Get procedure's execution, the input pointer would be left at the start of the field which begins "8.99".

For complex and long complex types the field width is computed from the value (2*w)+3 if a width is specified, or otherwise five characters. A single complex quantity is decoded from this region, all of which must be read from a single input record. Either of the formats allowed in free format complex input is valid.

For example:

```
complex C, D;
Get(3, "2E5", C, D);
```

If the input record was:

```
3.4+6.7iÞÞÞÞÞ-7.8-8.9i
```

then the value of the variables would be:

```
C = 3.4+6.7I
D = -7.8-8.9I
```

Similarly, if the input record was:

```
(3.456,7.892)(-6.7,2.3)
```

then the values of the variables would be :

```
C = 3.456+7.892I
D = -6.7+2.3I
```

"H" Format

This format is provided for use mainly with output procedures. If encountered during format directed input, the input record pointer is advanced one character position and the character specified after the format code is ignored.

"I" and "J" Formats

These two formats are used to input integer values into the next item in the <get-list>. While they each perform different functions when used as output formats, both cause the same effect during an input operation. Simple types integer, logical and bits are valid.

For all simple types the field width fetched from the input record is that specified with the format code, or else one byte. The same provisions apply as for "D", "E", and "F" formats, namely the field fetched must come entirely from one input record. A new record will be fetched if necessary.

Algol W will attempt to decode a single integer value from within this field in the same form as that expected in free format input. An error condition is recognized if more than one data item is found within this field.

If the <get-item> is of simple type bits, it will be set to the bit pattern corresponding to the internal representation of the integer value input - see Appendix J.

If the <get-item> is of simple type logical, an integer value of zero will cause it to be set to 'false'. Any other value will cause it to be set to 'true'.

"L" Format

This format is used to input logical values into the next item in the <get-list>. Simple types integer, logical and bits are valid.

For all simple types the field width fetched from the input record is that specified with the format code, or else one byte. The same provisions apply as for "D", "E", and "F" formats, namely the field fetched must come entirely from one input record. A new record will be fetched if necessary.

Algol W will attempt to decode a single logical value from within this field in the same form as that expected in free format input, that is "TRUE" or "FALSE", or any abbreviation down to "T" or "F", in either upper- or lowercase. An error condition is recognized if more than one data item is found within this field.

If the simple type is integer or bits then "FALSE" will send 0 or #0 and "TRUE" will send 1 or #1 respectively.

Example:

```
logical L1, L2, L3;
integer I;
bits B;
Get(2, "5L1", L1, L2, L3, I, B);
```

If the input data record is :

```
TTFTF
```

then L1 and L2 will be set to 'true', L3 to 'false', I to 1 and B to #0.

"T" Format

"T" format is used to position the internal pointer at a particular column in the physical input record ready for the next input operation. The field width given with the format specifies the column position to which the pointer should be set. If this width is omitted, column 1 is assumed. Three actions are possible.

(1) If the pointer is at a lower numbered position in the input record than that specified in the "T" format, it is moved to the requested position.

(2) If the pointer's current position is that specified by the "T" format, no action is taken.

(3) If the pointer's current position is beyond that specified by the "T" format, the remainder of the record is discarded and a new one is fetched. The pointer is then moved to the specified position.

Example:

```
integer I, J, K;
Get(2, "T4,2I3,T4,I4", I, J, K);
```

Suppose two consecutive physical input records read:

```
XY␣123456␣␣789
ZZ␣98765
```

The "T4" format will cause the following "I3" format to start at the fourth character position of the first input record, which is "1". I will be set to the value 123 and J to 456. The next "T4" format requests inspection of column 4 again. Since the input pointer for this record is beyond column 4, the second record is fetched and the pointer is set to column 4 within this record. K is then assigned the value "9876".

"X" Format

"X" format is used to skip character positions within an input record. It has one peculiarity: the width may be specified either before or after the "X", but not both. The value of this constant, or one if it is omitted, indicates the number of character positions which are to be skipped. If, while it is skipping characters, "X" format encounters the end of a physical input record, a new one will be fetched as many times as is necessary to complete the operation.

Example:

```
integer I, J, K;
Get(Input, "3(2X,I3)", I, J, K);
```

In this example a format group is repeated three times. The width of the group is 5 character positions, of which the first two are ignored under the control of the "2X" format and the last three are to be decoded as an integer. If the physical input record is:

```
12345␣␣678␣␣90123
```

then I will be assigned the value 345, J becomes 678 and K 901. Note that the initial characters "12" are ignored under the control of the "X" format.

## "Z" Format

This format is used to read hexadecimal characters into the next item in the <get-list>. The item may be of any simple type except reference. In general this format is intended for re-inputting data which has been output using "Z" format and, all other things being unchanged, the same "Z" format item which wrote the data out will read it in again.

When preparing data for input with this format, all hexadecimal characters which form a particular value must be included. It is not possible to omit any sequences of hexadecimal digits, or any spaces from data for string values.

The general form of the format item is "Z*w*.*d*". For all valid data types except string, the values of *w*.*d* have no significance. The effect of *w*.*d* is described in the section about format directed output using "Z" format.

The number of characters read in is twice the implied length of the <put-item>. A table of the implied lengths of different data types is given in the section describing "B" format.

Example:

```
integer I;
Get(Input, "Z", I);
```

to place the value 59 in the variable I the data record must be:

```
0000003B
```

Note that none of the leading zeros may be omitted.

## FORMAT DIRECTED OUTPUT

The following sections describe the action of each format in output operations.

### "/" Format

A slash symbol encountered in the format string causes the processing of the current output record to be terminated and this record to be written out. This format therefore starts a new line. Several slashes in a group will cause several output lines to be generated. Where a slash would cause an empty output line to be produced, Algol W produces an output record containing a single space. So a format string of "///" behaves as if it was "/X/X/".

Example:

```
Put(Print, "'Heading'//3X,'Index'");
```

This statement causes a new output record to begin on the PRINT stream, MTS SPRINT, and produces the following three lines of output:

```
Heading
ø
øøøIndex
```

The output pointer is left just after the "x" in the record containing the string "Index". The second line, containing only one blank, is caused by the two consecutive slashes.

### Literal String Format

This format is used to send a character string contained within the format string to the current output record. No <put-list> data item is processed. The literal string may be any sequence of characters enclosed in prime (') delimiters. If a prime is required as one of the characters within the literal string, two primes must be given.

Example:

```
Put(Print, "X,'Blacksmith''s anvil'");
```

would produce the physical output record:

```
øBlacksmith'søanvil
```

Note the contraction of the two primes to one in the output.

Other examples of the use of literal strings can be seen in the previous section describing the slash format.

## "A" Format

"A" format is used to output characters from the next item in the <put-list>. The only valid simple type for the <put-item> is string.

The number of characters sent to the output record is determined by the field width specified after the format code. If this is zero or omitted the implied length of the <put-item> is used instead. Where the field width is specified, its value should be between 1 and 256. If more characters are specified than are available in the <put-item>, then those available are output and padded on the right with blanks. Conversely, if the <put-item> contains more characters than the field width specifies, only the required number of characters is output from the string, starting with the leftmost character. Effectively the string is truncated on the right as it is output. If, when characters are being written to the output record, the current record becomes full, then it will be forced out and a new record started as many times as is necessary to output all the characters.

Example:

```
string(8) S, T;
S := "ABCDEFGH";
T := "0123WXYZ";
Put(Print, "A8,2X,A4", S, T);
```

In the above example eight characters are written to a new output record on the PRINT stream from string S. Next, two spaces are output under the control of the "2X" format code. Finally the "A4" format code causes only the first four characters of string T to be output. The output record will look like this:

```
ABCDEFGHbb0123
```

The output record pointer will be left just after the final "3".

## "B" Format

"B" format is used to output unedited bytes from the next data item in the <put-list>. Any simple type other than reference is valid.

For the output of strings the behavior of "B" format is almost identical to that of "A" format; the only difference being that, if any padding is necessary on the right, the pad characters will be hexadecimal 00 rather than spaces.

For other simple types the number of characters to be output is determined by the internal length implied by the <put-item>. These lengths are:

| | |
|---|---|
| integer | 4 |
| real | 4 |
| long real | 8 |
| complex | 8 |
| long complex | 16 |
| logical | 1 |
| bits | 4 |

For these simple types any width specified with the format is ignored: the lengths given in the above table are used instead. If the current physical output record has insufficient bytes remaining to contain the next output field, it is forced out and a new record is started. The entire output field is then written to this starting in column 1. Only string fields may be broken across several output records.

"B" format is designed to allow internal data to be output without editing, so that it may be re-input subsequently without any loss of accuracy due to rounding or other causes. Note that the internal forms of numerical and logical values so output may not be readily interpreted unless inspected in a hexadecimal dump. Number representation is discussed in Appendix J.

## "D" and "E" Formats

These two formats are used to output floating point values from the next item in the <put-list>. They have an identical action and exist only to give compatibility with Fortran conventions. They cause a floating point value to be output as a mantissa and exponent. The format code takes the form E*w*.*d* or D*w*.*d* where *w* specifies the width of the field to be output and *d* the number of decimal places to be included in the mantissa. Simple types real, long real, complex and long complex are valid.

For real and long real types, the field width specified, or otherwise one byte, will be written to the physical output record. If the current record does not have a field of this width remaining, it is forced out and a new one is started. The whole field is then written to this record starting in column 1.

Example:

```
real A;
A := 908.7348;
Puton(Print, "E14.3", A);
```

The output record will contain:

```
␣␣␣␣9.087'+02
```

The code "E14.3" has caused a field of 14 characters to be written out. This contains the value of A with an exponent of 10 to the power of two and three decimal places as specified by the format code.

Note that the specification of only three decimal places has caused the number to be rounded.

For complex and long complex types the field width is computed from the value (2*w)+3 if a width is specified, or otherwise five characters. A single complex quantity is decoded and output in this field, which will all be on a single output record. A new line will be forced if necessary.

Example:

```
complex C, D;
C := 3.4+6.7i;
D := -7.8-8.9i;
Puton(Print, "2E9.1", C, D);
```

will produce in the output record:

```
(␣␣3.4'+00,␣␣6.7'+00)(␣-7.8'+00,␣-8.9'+00)
```

"F" Format

This format is used to output floating point values from the next item in the <put-list> as a decimal number. The format code takes the form Fw.d where w specifies the width of the field to be output and d the number of decimal places to be included. Simple types real, long real, complex and long complex are valid.

For real and long real types, the field width specified, or otherwise one byte, will be written to the physical output record. If the current record does not have a field of this width remaining, it is forced out and a new one is started. The whole field is then written to this record starting in column 1.

Example:

```
real A;
A := 908.7348;
Puton(Print, "F14.3", A);
```

The output record will contain:

```
␣␣␣␣␣␣␣908.735
```

The code "F14.3" has caused a field of 14 characters to be written out. This contains the value of A with three decimal places as specified by the format code. The field has been padded with blanks on the left to give the requested width.

Note that, once again, the specification of three decimal places has caused the value of A to be rounded as it is output, but not to the same extent as with "E" format in the previous section, as, in this case, more significant digits have been printed.

For complex and long complex types the field width is computed from the value (2*w)+3 if a width is specified, or otherwise five characters. A single complex quantity is decoded and output in this field, which will all be on a single output record. A new line will be forced if necessary.

Example:

```
complex C, D;
C := 3.4+6.7i;
D := -7.8-8.9i;
Puton(Print, "2F9.3", C, D);
```

will produce in the output record:

```
(␣␣␣3.400,␣␣␣6.700)(␣␣-7.800,␣␣-8.900)
```

"H" Format

This format causes the single character which follows the format code to be written to the physical output record. It is particularly useful for generating carriage control characters for those output devices which need them.

Example:

```
Put(Print, "H0,'Heading'");
```

produces the new output record:

```
0Heading
```

where the character zero in column 1 will cause the line printer to skip two new lines when printing.

"I" Format

This format is used to output integer values from the next item in the <put-list>. Any integer value output using this format will be right justified in the field width specified. (Contrast this with the action of "J" format, which is described in the following section.) Simple types integer, logical and bits are valid.

For all simple types the field width written to the output record is that specified with the format code, or else one byte. The same provisions apply as for "D", "E", and "F" formats, namely the field must be written entirely to one output record. A new record will be started if necessary.

If the <put-item> is of simple type bits, the value output will be the integer corresponding to the bits value - see Appendix J.

If the <put-item> is of simple type logical, an integer value of 0 or 1 will be output according to whether the item has the value 'false' or 'true' respectively.

For example, the code:

```
integer I;
I := 4059;
Puton(Print, "I6", I);
```

will produce:

```
␣␣4059
```

in the output record.

Note that if the specified field width is too small to contain the <put-item>, a larger field width is assumed temporarily so that the complete integer may be output.

"J" Format

This format is used to output integer values from the next item in the <put-list>. Any integer value output using this format will be left justified in the field width specified. (Contrast this with the action of "I" format, which is described in the previous section.) Simple types integer, logical and bits are valid.

The fields specified by this format code are output in the same manner as that specified for "I" format in the previous section.

For example, the code:

```
integer I;
I := 4059;
Puton(Print, "J6", I);
```

will produce:

```
4059¢¢¢
```

in the output record.

"L" Format

This format is used to output logical values from the next item in the <put-list>. Simple types integer, logical and bits are valid.

For all simple types the field width written to the output record is that specified with the format code, or else one byte. The same provisions apply as for "D", "E", and "F" formats, namely the field must be written entirely to one output record. A new record will be started if necessary.

The word "FALSE" or "TRUE" will be printed in the output record according to the value of the <put-item> but regardless of its data type. If the field width specified is less then the length of the word, it will be truncated as necessary down to "F" or "T", respectively.

For example, the code:

```
logical L1, L2;
L1 := false;
L2 := true;
Puton(Print, "L6,L1", L1, L2);
```

will produce:

```
 FALSE¢T
```

in the output record.

"T" Format

"T" format is used to position the internal pointer at a particular column in the physical output record ready for the next output operation. The field width given with the format specifies the column position to which the pointer should be set. Three actions are possible.

(1) If the pointer is at a lower numbered position in the output record than that specified in the "T" format, it is moved to the requested position. The intervening character positions are filled with blanks.

(2) If the pointer's current position is that specified by the "T" format, no action is taken.

(3) If the pointer's current position is beyond that specified by the "T" format, the record is forced out and a new one is started. The pointer is then moved to the specified position in this new record, again padding with blanks.

Example:

```
Puton(Print, "T5,'NAME',T7,'TWO'");
```

will produce:

```
␣␣␣␣NAME
␣␣␣␣␣␣TWO
```

In general the output record pointer cannot move backwards (but see description of "T0" below) so when "T7" is encountered in the format code, the current physical output record is forced out and a new one is started.

"T1" is equivalent to slash, "/", unless the pointer is already at the start of a new physical output record, in which case no action is taken.

"T0" is a special use of the "T" format which causes the output pointer to be set immediately after the last non-blank character in the current physical output record. "T", with no column specified, is equivalent to "T0".

Example:

```
string (17) Data_Set_Name;
string (8) Root, Type;
Root := "SIMRTN";
Type := "SA";
Putstring(Data_Set_Name, "A,T,H.,A", Root, Type);
```

The effect of the first "A" format is to put into Data_Set_Name:

```
SIMRTNbb
```

The final value of Data_Set_Name is:

```
SIMRTN.SAbbbbbbbb
```

Note that the "T" format has backspaced over the trailing blanks inserted by the previous "A" format, thereby allowing the required dataset name to be built up in the string without any embedded blanks.

A more complex development of this example could be used to build a Control statement procedure argument to control an MTS magnetic tape.

## "X" Format

"X" format is used to insert space characters within an output record. It has one peculiarity: the width may be specified either before or after the "X", but not both. The value of this constant, or one if it is omitted, indicates the number of spaces which are to be inserted. If, while it is inserting characters, "X" format encounters the end of a physical output record, a new one will be started as many times as is necessary to complete the operation.

Example:

```
integer I, J, K;
I := 1234;
J := 678;
K := 901;
Puton(Print, "3(2X,I4)", I, J, K);
```

In this example a format group is repeated three times. The width of the group is six character positions of which the first two are spaces under the control of the "2X" format and an integer is to be output into the last four. The physical output record will contain:

```
bb1234bbb678bbb901
```

Each integer field is prefixed by two spaces.

## "Z" Format

This format is used to output values from the next item in the <put-list> in a hexadecimal format in which each byte is represented as two hexadecimal digits. All simple types except reference are valid.

The format code takes the general form "Z*w*.*d*", where the values of *w* and *d* have significance only if the <put-item> is of type string. In this case they control the number of characters output and their spacing as described later in this section.

If the <put-item> is not of simple type string, the values of *w*.*d* are ignored. The number of hexadecimal digits output from the <put-item> is calculated as twice the implied length of the item. A table of the implied lengths of different data types was given in the section describing "B" format.

If there is insufficient space remaining in the current output record to contain the entire item, the record is forced out and a new one is started, the item being written to this starting in column one.

Example:

```
integer I;
I := 59;
Put(Print, "Z", I):
```

will produce:

```
0000003B
```

If the <put-item> is of simple type string, the value of *w* will be used as the number of hexadecimal digits to be output. If the value given is an odd number, one is added before it is used. If the value of *w* is shorter than twice the implied length of the <put-list> item then its contents will be truncated on output. Conversely, where it is greater, the value of the <put-item> is padded on the right with the characters "00".

If *w* is omitted or zero, the number of characters output from the <put-item> is twice its implied length.

The maximum value which *w* may have is 512.

The value of *d* controls the number of spaces inserted between each block of eight hexadecimal characters in the representation of a particular item in the output. *d* may have a value between the default of 0 and 8.

Each field of 8 hexadecimal characters plus *d* spaces must fit entirely onto the remainder of the current output record. Otherwise this is forced out and a new record started.

Example:

```
string(8) S;
S := "ABCDEFGH";
Put(Print, "Z16.2", S);
```

will produce:

```
     C1C2C3C4bbC5C6C7C8
```

"R" - THE DATA DRIVEN REPLICATION FACTOR

In the previous descriptions of format directed output codes, the replication factor r has always been an integer constant. For output only, certain format codes may specify that the replication factor is taken from the next item in the <put-list>.

The format codes allowable are:

| | |
|---|---|
| X | Spaces |
| H | Character insertion |
| T | Output tab |
| '...' | Character string insertion |

The letter "R" in place of the normal integer constant specifies that the replication factor is to be taken from the next item in the <put-list>. This <put-item> must be an integer expression.

For example, a skip of five spaces would normally be coded as:

```
Puton(Wtr, "5X");
```

If the number of spaces skipped is a program variable, the example becomes something like:

```
integer Nspaces;
 .
Nspaces := ...
Puton(Wtr, "RX", Nspaces);
```

In this case the number of spaces output is supplied by the value of the integer variable Nspaces. The "X" format otherwise behaves as usual. The Puton statement here is equivalent to:

```
for I := 1 until Nspaces do Puton(Wtr, "X");
```

but use of the "R" factor is more efficient, since fewer statements are executed.

Consider the following program:

```
begin
   integer Aline, Magnitude;
   for Line := -5 until 5 do
   begin
      Aline := abs Line;
      Magnitude := (5 - Aline)*4 + 1;
      Put(Wtr, "X,I5,2X", Magnitude);
      Puton(Wtr, "RH*,8X,H#", Magnitude);
   end Print_Loop;
end.
```

The last Puton statement uses the "R" form of "H" format to output a line of asterisks whose length is determined by an algorithm within the program code. When run, this program produces the following output:

```
 1  *        #
 5  *****        #
 9  *********        #
13  *************        #
17  *****************        #
21  ***********************        #
17  *****************        #
13  *************        #
 9  *********        #
 5  *****        #
 1  *        #
```

Note that the number of asterisks output is given by the tabulated values of Magnitude at the left hand side. The effect of the trailing "8X,H#" shows that the "RH" format left the output pointer just after the last asterisk output.

## SAMPLE PROGRAM USING FORMAT DIRECTED OUTPUT

The following program tabulates the values of several function expressions at an interval of 0.2; the initial and and final values are read in from the basic input stream.

```
begin
   real Start, Finish, Current;
   integer Istart, Ifinish;
   Read(Start, Finish);
   Istart := Round(Start*5) * 2;
   Ifinish := Round(Finish*5) * 2;

   Put(Wtr, "H-,A", "Function Table");
   Put(Wtr, "H0,'Initial value = ',F5.1", Istart/10);
   Put(Wtr, "X,'Final value   = ',F5.1", Ifinish/10);
   Put(Wtr, "H-,5X,A8,3(2X,A10),2X,A12//",
       "Value", "Sqrt", "Log-e", "Log-10", "Exp**3");

   for Index := Istart step 2 until Ifinish do
   begin
      Current := Index / 10;
      Put(Wtr, "X,F8.1,3(2X,F10.3),2X,E12.2",
          Current, Sqrt(Current), Ln(Current),
          Log(Current), Exp(Current) ** 3)
   end Tabulate_Loop;
end.
```

The initial group of four Put statements write the heading lines, and the final Put within the 'for' loop block produces the actual table. The purpose of the Round expressions in the assignments of the variables Istart and Ifinish is to produce starting values at the nearest 0.2 interval.

Note that the format string of the Put statement which produces the heading for each column of the table is in fact derived from the format string of the statement which does the actual tabulation. A simpler format string could have been used to produce this heading, but this method reduces the labor necessary to get the spacing correct.

If the data values input are:

```
1.52   3.34
```

then the following output is produced:

```
     Function Table

     Initial value =   1.6
     Final value   =   3.4
```

| Value | Sqrt | Log-e | Log-10 | Exp**3 |
|---|---|---|---|---|
| 1.6 | 1.265 | 0.470 | 0.204 | 1.22'+02 |
| 1.8 | 1.342 | 0.588 | 0.255 | 2.21'+02 |
| 2.0 | 1.414 | 0.693 | 0.301 | 4.03'+02 |
| 2.2 | 1.483 | 0.788 | 0.342 | 7.35'+02 |
| 2.4 | 1.549 | 0.875 | 0.380 | 1.34'+03 |
| 2.6 | 1.612 | 0.956 | 0.415 | 2.44'+03 |
| 2.8 | 1.673 | 1.030 | 0.447 | 4.45'+03 |
| 3.0 | 1.732 | 1.099 | 0.477 | 8.10'+03 |
| 3.2 | 1.789 | 1.163 | 0.505 | 1.48'+04 |
| 3.4 | 1.844 | 1.224 | 0.531 | 2.69'+04 |

# EXTERNAL LINKAGES

This section describes the following Algol W facilities:

(1) Calling Algol W procedures which have been compiled separately from a main Algol W program.

(2) Calling subroutines which use the IBM O/S Type I linkage - sometimes referred to as the S-type linkage. This is used by all FORTRAN subroutines and very often by Assembler programs.

(3) Calling Algol W procedures from programs written in languages other than Algol W (e.g., FORTRAN, Assembler) using the O/S Type I linkage.

## CALLING ALGOL W PROCEDURES

### Coding External Algol W Procedures

An Algol W procedure can stand alone as a program provided that it satisfies the following restrictions:

(1) It cannot reference any global identifiers, except those in the "supplied" block containing all predeclared identifiers and thus accessible to any Algol W program. Except for these predeclared ones, identifiers that are not declared within the procedure may not be accessed.

(2) Declarations of record classes (and thus of reference quantities) are subject to special rules. They should be avoided in externally defined procedures.

(3) If the procedure is to be called from a non-Algol W routine (e.g., FORTRAN, Assembler) using the O/S Type I linkage then all array parameters must be singly dimensioned and the procedure cannot be a string function procedure.

Stand alone Algol W procedures can be compiled in a normal way with the DECK option specified - see the section "Algol W Programmer's Guide." An example of a procedure which may be called externally is:

```
integer procedure Power(integer value I, Exp);
begin
   integer Res;  Res := 1;
   for K := 1 until Exp do Res := Res * I;
   Res
end.
```

Note the terminating period (.) which follows a procedure which is being compiled separately.

In the environment known to the operating system loader such precompiled procedures are known by the name of their entry point. This name is known as the external symbol definition name or ESDname. ESDnames are always eight characters in length. The ESDname for the procedure given in the above example would be "POWERbbb", where b is a single blank character. These names are formed from the capitalized procedure name either by truncating it to eight characters or by padding to this length with blanks.

Note however that if several precompiled procedures are to be used together then the first five characters of each name should be unique. When the compiler is processing nontrivial blocks within procedures, it will generate additional loader modules whose ESDnames consist of the first five characters of the capitalized procedure name followed by suffix characters to distinguish the individual module. Also it is good practice to avoid procedure names beginning with the characters "AW" since the Algol W system modules all begin with this prefix. Following these two rules will avoid many difficulties caused by loader module name clashes, which, in certain circumstances, may cause control to be transferred to a routine other than the one intended.

## Calling Precompiled Procedures from Algol W

In order to call a precompiled Algol W procedure from an Algol W main program or procedure, it is necessary to declare the heading of the procedure, followed by an external reference to it.

An external reference for an Algol W precompiled procedure consists of the reserved word 'algol' followed by a string constant containing the name of the precompiled procedure to be called. Such a phrase stands as a procedure body in a procedure declaration and establishes the connection between the Algol W program or procedure and the precompiled procedure which is being called. The string following the reserved word 'algol' is capitalized by the compiler when forming the relevant external reference.

Externally defined Algol W procedures may then be called in exactly the same way as a procedure nested within the main program.

Example:

```
     begin

        integer procedure Power(integer value I, Exp);
           algol "POWER";

        integer A, B, C;
        Read(A, B);
        C := Power(A, B);
        Write("Result is", C)
     end.
```

This example shows a program calling the precompiled procedure given in the previous subsection. Here a function procedure is being declared as an external function procedure, and then called. The description of the external 'algol' linkage applies equally well to proper procedures.

## Calling Precompiled Procedures from Outside Algol W

Precompiled Algol W procedures may be called from programs written in languages other than Algol W by using the O/S Type I calling conventions. FORTRAN, Assembler, and many other languages use the O/S Type I calling conventions. No special processing of the Algol W routine is required. The ESDname of the precompiled Algol W procedure consists of the first eight characters of the Algol W procedure name (padded with blanks if necessary).

Only the outermost precompiled procedure may be called from outside Algol W using the O/S Type I linkage conventions. Internal Algol W procedures may be called using the Link predefined procedure (see the section "Link").

If the Algol W procedure to be called requires parameters then exactly that number of parameters must be passed to the Algol W procedure or a run-time error will occur; no variable length parameter lists are allowed. Any simple type or singly dimensioned array may be passed as a parameter to the Algol W procedure. The following restrictions are imposed on parameters of precompiled Algol W procedures that are to be called from a non-Algol W routine:

(1) No formal procedure parameters (i.e. parameters which are procedures) may be given.

(2) Logical variables are the equivalent of FORTRAN's LOGICAL*1. If the FORTRAN routine uses LOGICAL or LOGICAL*4 the corresponding Algol W parameter should be declared integer or bits.

(3) Arrays may only have one dimension and the lowest subscript will always be one.

It is not possible for Algol W to do any kind of type checking of the parameters that are passed to the Algol W procedure. Therefore, the programmer must take care to ensure that the types of the parameters match exactly. In particular, when passing strings to the Algol W procedure be sure that the string lengths match. No padding or truncation of the strings is done.

Any of the four Algol W parameter passing conventions (value, result, value-result, and name) may be used in an Algol W procedures called from a non-Algol W program. Call by reference is the standard parameter passing convention used with the O/S Type I. Call by reference is not available explicitly in Algol W. However, call by name in Algol W is identical to call by reference when used with the O/S Type I linkage convention. Because of several unpleasant side-effects of call by name, it is highly recommended that call by value, result, or value result be used instead of call by name.

For example, the Algol W procedure:

```
        procedure Sumsq(long real value A,B;
                        long real result C);
        begin
           C := A ** 2 + B ** 2;
        end.
```

Can be called in the following way from a FORTRAN program:

```
            REAL*8 A1, A2, SUMOF
            READ(5,1000) A1,A2
      C ***   CALL THE ALGOL W PROCEDURE
            CALL SUMSQ(A1, A2, SUMOF)
            WRITE(6,1001) SUMOF
            STOP
       1000 FORMAT(F7.2, 2X, F7.2)
       1001 FORMAT(' RESULT IS', F7.2)
            END
```

## Explicitly Initializing the Algol W Environment

When an Algol W procedure is called from outside Algol W it first must access the Algol W run-time environment. If this environment does not exist, then the Algol W procedure must allocate and initialize the environment and set any default values (e.g., the size of the environment, whether a dump is to be produced if an error occurs, etc.). Initializing and accessing the environment is normally done automatically by the Algol W procedure when it is called and most users need never even know that the environment exists. However, some users may wish to alter the default settings of the run-time environment. The ALWBEG routine is provided to allow the user to explicitly initialize the environment and to supply a string of run-time parameters to be used

during this initialization. Note that this routine can only initialize the environment if it has not already been initialized. Therefore, it never makes sense to call this routine from Algol W or after a call to an Algol W routine has been made. The following is a description of ALWBEG:

ALWBEG

Purpose: To initialize the Algol W run-time environment and to supply run-time parameters to Algol W.

Location: Resident system

Calling Sequences:

Assembly: CALL ALWBEG,(parstring)

FORTRAN: CALL ALWBEG(parstring,&rc4,&rc8)

Parameters:

parstring is the location of a halfword length followed by a string of Algol W run-time parameters.

rc4,rc8 are statement labels to transfer to if a nonzero return code is encountered.

Return Codes:

0 Successful return.

4 The Algol W run-time environment already exists; the parstring was not processed.

8 Errors were detected in the parstring.

Description: The ALWBEG subroutine initializes the Algol W run-time environment for use by Algol W routines that will presumably be called later. It is usually not necessary for a user to call ALWBEG since the Algol W run-time environment is automatically initialized the first time an Algol W procedure is called. However, ALWBEG also allows the user to specify a string containing containing Algol W run-time parameters.

Examples: Assembly:

```
         CALL ALWBEG,(PARSTR)
           .
           .
PARSTR   DC  H'9'
         DC  CL9'SIZE=100P'
```

```
FORTRAN:  INTEGER2 PARSTR(4)
          DATA PARSTR/6,'NODUMP'/
                .
                .
           CALL ALWBEG(PARSTR)
```

## Deallocating the Algol W Environment

Most users never need to deallocate the Algol W environment. However, some users may find that explicitly deallocating the Algol W environment will reduce the cost of running their programs by releasing the virtual memory used by the program and thereby reducing the VM charges. The environment may be deallocated by calling the ALWEND procedure. If an Algol W procedure is called after the environment has been deallocated using ALWEND then the environment will be reinitialized. Repeatedly initializing and deallocating the Algol W environment may be expensive. It is therefore recommended that ALWEND be called only after all processing by Algol W routines is completed.

It is possible for ALWEND to be called from a routine which was in turn called by an Algol W routine. If this happens then it is not possible to deallocate the environment since the Algol W routine will need the environment when the non-Algol W routine returns to it.

The following is a description of ALWEND:

### ALWEND

Purpose: To shut down the Algol W run-time environment.

Location: Resident system

Calling Sequences:

Assembly: CALL ALWEND

FORTRAN: CALL ALWEND(&rc4,&rc8)

Parameters:

rc4,rc8 are statement labels to transfer to if a nonzero return code is encountered.

Return Codes:

0 Successful return.

4 Algol W routines are active.

8 No ALgol W run-time environment exists.

Description: The ALWEND subroutine shuts down the Algol W run-time environment. This involves flushing all Algol W output buffers and releasing all storage associated with Algol W. Note that ALWEND cannot shut down the environment if the calling subroutine was itself called from an Algol W procedure. If this were done then catastrophic errors would occur when the calling subroutine returned to the active Algol W procedure.

Calling ALWEND may yield a significant reduction in the virtual memory charges accrued by a program. However, calling ALWEND and then calling an Algol W procedure will cause the Algol W run-time environment to be reinitialized. This overhead associated with reinitializing the environment may be considerable if this is done repeatedly.

Examples: Assembly: CALL ALWEND

FORTRAN: CALL ALWEND

Link - Procedure Call Back from an External Subroutine

Certain FORTRAN subroutines may require as a parameter either another subroutine or a function. FORTRAN expects that this subroutine or function will be callable by the normal O/S Type I linkage convention. FORTRAN subroutines and externally compiled Algol W procedures use this convention, but internal Algol W procedures (contained in a block or procedure) do not. Subroutine calling conventions are fully discussed in Appendix K.

The following partial FORTRAN subroutine shows the kind of call which may be required.

```
      SUBROUTINE FSUB(X,Y,N,Q)
C
C  FOURTH PARAMETER, Q, IS CALLED AS A
C  FUNCTION SUBPROGRAM DURING EXECUTION
C
      REAL*4 X(Y),Y(N)
       .
       .
      A = Q(X(I))
       .
       .
      RETURN
      END
```

In this example the fourth parameter of the subroutine FSUB, Q, is intended to be a supplied function. The call of this function in the line:

```
A = Q(X(I))
```

shows that this function is supplying as its only parameter a real value and returning a real result.

If the subroutine FSUB is to be called from Algol W then the fourth parameter must somehow supply a FORTRAN callable function of this type. There are two possible ways in which this may be done:

(1) If the function to be supplied is coded in FORTRAN, or is otherwise separately provided and uses the O/S Type I linkage, then a reference to this external routine can be provided using the predeclared function External which is described in the section "Miscellaneous Topics."

(2) If, on the other hand, it is desired to supply a main program Algol W procedure as the function Q, then a reference must be supplied to this procedure so that a call may be set up to it using the FORTRAN linkage convention. This is achieved using the Link predeclared procedure described in the remainder of this section.

Link may only be specified when used directly as an argument to the Call predeclared procedure.

The general form of the function call is:

```
Link(<procedure-designator>)
```

where <procedure-designator> is a string constant containing only the name of the main program Algol W procedure to be supplied to a FORTRAN subroutine or function. It will therefore always appear in the source program text as a procedure identifier enclosed in quotes ("). The procedure so designated must be in scope when the Call statement is issued according to the normal rules of identifier scope within Algol W.

When an Algol W procedure is intended to be called from FORTRAN certain restrictions are imposed on its parameters:

(1) No formal procedure parameters (i.e. parameters which are procedures) may be given.

(2) Logical variables are the equivalent of FORTRAN's LOGICAL*1. If the FORTRAN routine uses LOGICAL or LOGICAL*4 the corresponding Algol W parameter should be declared integer or bits.

(3) Arrays may only have one dimension and the lowest subscript will always be one.

Using the previous example of the FORTRAN subroutine FSUB, the following Algol W program shows how an Algol W main code procedure could be supplied to it using Call and Link together.

```
begin

   real procedure Xfunction(real value Arg);
   begin
      real Res;
      Res = 0.5 + Sqrt(Arg);
      Res
   end Xfunction;

   real array X, Y(1::1000);
    .
    .
   Call("FSUB", X(1), Y(1), 1000, Link("Xfunction"));
    .
    .
end.
```

Note that, as described previously, the arrays X and Y are specified by giving their first elements as parameters. It is essential that the Link call is nested as a parameter to Call. This is because Algol W generates code at this point to translate the FORTRAN calling conventions correctly to those of Algol W in order to call Xfunction. When the called procedure finishes execution, the FORTRAN environment is restored in order to return control to the FORTRAN subroutine FSUB.

## CALLING FORTRAN, ASSEMBLER, AND RELATED SUBROUTINES

The facilities described in this section are provided to call routines which use the IBM O/S Type I subroutine linkage from Algol W programs. In the commonest form, known as the S-type (for storage location) call, parameters are supplied to the called routine as a serial list of machine addresses in main storage called a parameter list. A simpler but less commonly used form, known as the R-type (for register) call, supplies parameters as values contained in actual machine registers.

Full details of the calling conventions, together with details of how to code assembler routines to use with Algol W, will be found in Appendix K. The following subsections describe various methods of calling this kind of subroutine.

Call

Call is a predeclared procedure (like Write) which is used to set up a call to an external subroutine using the S-type linkage. It has a number of advantages over the older calling mechanism using the 'fortran' external reference:

(1) Subroutines to be called do not need to be declared before use.

(2) No type checking is done on the supplied parameters. Any simple type variable or expression may be given as a parameter.

The general form of the procedure call is either of:

```
Call(<esdname>)
Call(<esdname>, <parameter-list>)
```

where:

<esdname> is either a string constant of maximum length eight characters, an integer expression or a bits expression; and

<parameter-list> is a list of one or more expressions called parameters, each of which is separated from the next by a comma.

The <esdname> specifies the subroutine to be called. If it is a string constant, it should be given as a 1 to 8 character entry point name of the subroutine with no blanks specified. The characters in this string constant will be capitalized when forming the relevant external reference. If it is an integer or bits expression then the value of the expression is taken to be the address of the entry point of the subroutine. In both cases control would be passed to the entry point specified, in the manner described in Appendix K.

If the subroutine to be called requires parameters, then parameters 1 to n of the subroutine are given as arguments 2 to n+1 of the Call predeclared procedure. Any simple type variable or expression may be given as a parameter. If, however, a value is to be returned through a parameter, the name of the receiving variable should be given. This is very important as, if an expression or constant is given, the returned value will be lost. When in doubt supply a variable, presetting an initial value for the subroutine if required.

When supplying string variables or substring designators as receiving variables be sure that the length of the string is sufficient to contain the number of bytes which will be transmitted back by the subroutine to the Algol W program.

For all parameters sent by Call, an address is computed for the subroutine parameter list. Note that this means that for a string variable, S, the name of the variable, S, and the substring S(0|1) are identical when given as a parameter to Call. The substring length (of

one) in no way prevents a called subroutine from storing data into subsequent bytes in the string, such as that designated by S(1|1). System input subroutines are particularly dangerous in this respect - be sure that the length of a supplied input buffer is sufficient to contain the characters which will be transmitted by the routine.

For example, the FORTRAN subroutine:

```
      SUBROUTINE SUMSQ(A,B,C)
C
C  SUM THE SQUARES OF THE FIRST TWO PARAMETERS
C  AND RETURN THE RESULT IN THE THIRD
C
      REAL*8 A,B,C
      C = A**2 + B**2
      RETURN
      END
```

can be called in the following way:

```
begin
   long real A1, A2, Sum_Of_Squares;
   Read(A1, A2);
   comment call the FORTRAN subroutine;
   Call("SUMSQ", A1, A2, Sum_Of_Squares);
   Write("Result is", Sum_Of_Squares);
end.
```

## Literal Parameters using Call

When literal parameters, that is constants, are supplied to an external subroutine using Call, extreme care must be taken to ensure that Algol W can correctly decide the simple type of the literal. The following table shows a series of literals which are self-typing.

| Literal | Simple Type |
|---|---|
| 1 | integer |
| 1. | real |
| 1L | long real |
| 1+1I | complex |
| 1L+1IL | long complex |
| true | logical |
| "1" | string |
| #1 | bits |
| null | reference |

The points to note when inspecting the table are:

(1) a decimal point will coerce an integer literal into a real;

(2) a trailing "L" will coerce a numeric literal into a long real quantity;

(3) a trailing "I" will coerce a numeric literal into a complex quantity;

(4) logicals may take only two values, 'true' or 'false';

(5) string quantities must be enclosed in quotes;

(6) bits literals must be prefixed by a hash mark (#);

(7) there is only one reference literal, the pointer 'null', but this should never be needed when calling an external subroutine.

Because the machine representation of integer and floating point quantities is completely different, it is essential when supplying literal parameters of these types to distinguish carefully between an integer quantity, say 99, and the corresponding floating point values 99.0 and 99.0L.

The representation of numeric quantities on the machine is fully discussed in Appendix J.

The following example shows a series of subroutine calls where the parameters are literals.

```
begin
   Call("IGINIT");
   Call("IGBGNS", "FRED");
   Call("IGMA", 0.0, 0.667);
   Call("IGDA", 0.5, 0.3);
   Call("IGDA", -0.5, -0.3);
   Call("IGDA", 0.0, 0.667);
   Call("IGMA", 0.0, 0.0);
   Call("IGMA", 0.0, -0.667);
   Call("IGDA", 0.5, -0.3);
   Call("IGDA", -0.5, 0.3);
   Call("IGDA", 0.0, -0.667);
   Call("IGMA", 0.0, 0.0);
   Call("IGENDS", "FRED");
   Call("IGDRON", "TERM");
end.
```

In this example the programmer has been careful always to specify the correct type of literal parameter required by the subroutine. The subroutines called in this example are part of the MTS integrated graphics package and the program draws two overlapping triangles.

Arrays as Parameters using Call

Some special problems occur when an array is to be passed as a parameter to an external subroutine using the Call mechanism described in the previous section. Because Call is a predeclared procedure (like Write) its parameters may only be one of the nine simple variable types: integer, real, long real, complex, long complex, logical, string, bits or reference. This list excludes arrays of simple variables and records. In practice there would be few occasions when record or reference parameters would be required for an external subroutine so the problem is reduced to the supply of arrays as parameters.

The solution to this problem is fairly easy but requires some thought by the programmer about the exact kind of parameters required by the called subroutine.

In FORTRAN and languages with similar subroutine calling conventions an array of variables consists of a number of individual variable cells which are stored adjacent to each other in main storage. They are stored in sequence so that the address of a particular element, and thereby its value, may be obtained readily from the element's subscript.

The technique of passing an array via Call is to pass the address of the first element of the array as a parameter. This is done by supplying as the parameter the subscripted variable required. This must be supplied alone as a parameter; it must not be part of a larger expression as this would cause Algol W to evaluate the expression and pass the address of the result.

Example:

```
begin
   long real array Data(1::100);
   long real Sum;
    .
    .
   Call("ARYSUM", Data(1), 100, Sum);
    .
end.
```

This example shows how an array Data is supplied to a subroutine ARYSUM by specifying its first element.

There are other points to note when passing arrays to FORTRAN. In FORTRAN arrays are specified as dimensioned variables. The lower subscript or subscripts of a FORTRAN dimensioned variable is always one. The consequences of this are as described below.

Single dimensioned arrays:

Since there is no equivalent in FORTRAN of the Algol W type of array declaration, in which both upper and lower subscripts are

specified, an Algol W programmer must be careful to supply the correct part of a vector to a FORTRAN subroutine. In the previous example, where the array lower bound is one and the whole of the array is being supplied to the subroutine, there is no problem. However, if the lower bound of the Algol W array is not one, care must be taken to provide the external subroutine with the required vector mapping. Example:

```
begin
   integer array Seq(-20::20);
    .
    .
   Call("SUBA", Seq(1));
   Call("SUBB", Seq(-20));
    .
end.
```

The first call, of subroutine SUBA, supplies an array in which the Algol W subscript mappings are identical, that is Seq(5) specifies the same element in both Algol W and the called FORTRAN subroutine SUBA. However elements -20 to zero of the array Seq are not available to SUBA.

In the second call, of subroutine SUBB, the parameter specifies the start of the Algol W array Seq. Since FORTRAN numbers its arrays from one upwards this will change the mapping of the array elements when SUBB accesses it as a parameter. Element -20 in Algol W will become element one in the FORTRAN subroutine SUBB and similarly the rest of the array elements will seem to have 21 added to their subscripts, so that the elements in FORTRAN run from 1 to 41 instead of from -20 to 20. Note that in this second example the whole of the array is available to SUBB.

Multiple dimensioned arrays:

Extreme care must be taken when passing arrays with more than one dimension. Both Algol W and FORTRAN store array elements in column major order, that is with the leftmost subscript varying most rapidly, so no problems of element mapping will occur for identical arrays.

There is no consistency check within FORTRAN which will detect the fact that a passed array has a different number of elements per column than the subroutine expects. Serious difficulties will be encountered if attempts are made to pass multiple dimensioned arrays of different dimensions to those expected by the called subroutine.

Multi-dimensioned arrays passed to FORTRAN subroutines should therefore be organized so that they are exactly the size required by the subroutine whether or not the subroutine is told the dimensions by parameters. It would be wise to have the Algol W array with a lower subscript of one for each dimension. Example:

```
begin
   integer N, M;
   read (N, M);
   begin
      long real array Ftndata(1::N, 1::M);
       .
       .
      Call("Q99XYZ", Ftndata(1,1), N, M);
       .
       .
   end;
end.
```

This example assumes that the called FORTRAN subroutine Q99XYZ has a calling sequence equivalent to a declaration of:

```
SUBROUTINE Q99XYZ(FTNDATA, N, M)
REAL*8 FTNDATA(N,M)
```

where FTNDATA is a REAL*8 array equivalent to the long real declaration of Ftndata in the Algol W program and N and M are INTEGER*4 variables equivalent to the integer variables within Algol W. In this case, it is assumed that N and M specify the first and second dimensions of the array as used by the FORTRAN subroutine. When a subroutine is specified from a library written in FORTRAN it is usually quite clear from the documentation when integer parameters specify the dimensions of an array parameter.

A partial array cannot be supplied to a FORTRAN subroutine by using the Algol W notation of an asterisk (*) as an array dimension when calling external subroutines via the Call predeclared procedure. This feature uses a special mechanism (called an array descriptor or dope vector) which has no parallel within FORTRAN.

## Subroutine Return Codes using Call

Many FORTRAN and Assembler subroutines supply an indication of their success or failure by an entity known as a return code. This is in fact a value left by the subroutine in machine general register 15. After control has been returned by the subroutine referenced in a Call statement the value of register 15 is stored in the predeclared Algol W integer variable R_Code.

It is an almost universal convention that a subroutine which has returned successfully will leave a return code of zero. Any other value indicates that an error of some kind has occurred and the subroutine could not complete its allotted task. Many subroutines return an error indication which is a multiple of four. In such subroutines the error condition may be processed easily on return to Algol W by the use of a Case statement.

When a subroutine is known to return an indication of success or failure in this way, the value of R_Code should always be checked by the program on return. Never assume success.

Example:

```
begin
   string(80) Filename;
   Read(Filename);
   Call("DESTROY", Filename);
   if R_Code = 0 then
      Write("Destroyed O.K. - ", Filename)
   else
      Write(case R_Code div 4 of (
            "Can't be destroyed",
            " ",
            "Does not exist",
            "Deadlock would result",
            "Access not allowed",
            "Parameter error",
            "Wait interrupted"))
end.
```

This program, intended to run in MTS, reads in a file name and then calls the system subroutine DESTROY to delete it from the system. DESTROY is a subroutine which returns one of a series of error codes which are multiples of four. Zero means success. The program deals with the error condition by using an expression containing R_Code as the subject of a Case clause.

## Obtaining Function Values using Call

Certain FORTRAN subroutines, called function subprograms, are the FORTRAN equivalent of Algol W function procedures. As such they return a value. Call, being a predeclared procedure, stands alone as a statement and cannot be used as the right hand part of a assignment statement as would be the case with a function procedure call. For this reason the Call mechanism always sets the values of certain Algol W predeclared variables to the returned values left by FORTRAN after the subroutine call. These variables are always set after Call. However they will only be meaningful if a function has been called which has set the required return value. The predeclared variables are:

R0

If a function returns an integer value it will be found in the predeclared integer variable R0 on return from the call. In the case of a logical value zero means false and one means true. In fact, R0 contains the value left in machine general register zero at the end of the subroutine call.

R_Float

Real and long real values are returned in a predeclared long real variable R_float. Note that, although this variable is of long real accuracy, for a real function the value will be no more accurate than an Algol W real. In fact, R_Float contains the value left in machine floating point register zero after the subroutine call.

R_Cmplx

Complex and long complex values are returned in the predeclared long complex variable R_Cmplx. The real part of R_Cmplx is the same storage area as the previously described long real variable R_Float. R_Cmplx contains the values left in the machine's floating point registers zero and two after the subroutine call, where a complex or long complex function result would be left.

R1

Certain subroutines return values in general register one. The value of the contents of this register are saved in the predeclared integer variable R1 after each call.

R01

In certain cases the returned values left in machine general registers zero or one or both are more usefully processed in Algol W as strings. For this reason the predeclared string(8) variable R01 is supplied. It specifies the same regions of Algol W system storage as the predeclared integers R0 and R1 but allows the values returned to be processed as strings. R01(0|4) is equivalent to R0 and R01(4|4) is equivalent to R1.

Further discussion of these variables will be found in the section on the alternate predeclared procedure Rcall.

## A Working Example using Call

The following illustrates how to call a Fortran subroutine with a rather complicated set of parameters. Assuming the Fortran subroutine is:

```
      SUBROUTINE MEAN(NUM,VALUES,AVRAGE)
C
C
C    This subroutine finds the average of NUM real numbers
C
C
C    NUM - the number of numbers to be averaged
C    VALUES - an array of NUM real numbers to be averaged
C    AVRAGE - the long real average of the numbers
C
      REAL VALUES(1)
      REAL*8 AVRAGE
C
      AVRAGE = 0.
      DO 10 I = 1, NUM
      AVRAGE = AVRAGE + VALUES(I)
 10   CONTINUE
C
      AVRAGE = AVRAGE / NUM
      RETURN
      END
```

The following Algol W program calls this Fortran subroutine:

```
begin
   comment this program reads in some numbers and prints
           out their average;

   integer Number_Numbers;

   comment read in the number of numbers to be averaged;
   Read(Number_Numbers);

   begin

      comment dynamically allocate the array of numbers;
      real array Numbers(1::Number_Numbers);

      long real Average;

      comment Read in the numbers;
      for I := 1 until Number_Numbers do
         Readon(Numbers(I));

      comment average them;
      Call("MEAN",Number_Numbers,Numbers(1),Average);

      Write("The average is ",Average);

   end
end.
```

Assuming the following input:

```
10
2.3  4.5 9.623 4.8
5.7  -2.5 0.0 100.123 27.5 9.2
```

The output would be:

```
The average is     16.1245983
```

Rcall

Certain system subroutines, while generally obeying the O/S Type I calling convention, do not use a standard S-type parameter list for communication between the calling and called routines. Instead they use values placed directly in machine general registers. In order to call such a subroutine from a high level language like Algol W, a method must be provided for loading values into machine registers before the call and retrieving the results, if any, on return. A predeclared procedure Rcall is provided to do this.

The general form of the procedure call is:

```
Rcall(<esdname>)
```

where <esdname> is as described in the section "Call" in this section.

Rcall will only set up machine general registers zero and one. When the subroutine is called the values of the predeclared integer variables R0 and R1 are loaded into the corresponding general registers zero and one. On return, the values in machine general registers zero, one, and 15 and floating point registers zero and two are returned as described in the previous two sections, that is the values are saved in the predeclared variables R0, R1, R_Code, R_Float, and R_Cmplx.

Where the parameters in registers zero or one or both are better treated as strings in Algol W they can be manipulated using the predeclared string(8) variable R01 as previously described.

Rcall is provided as a last resort when Call cannot handle the subroutine. It is expected that only very rarely will programs need to use Rcall. Those users who do need it may also find the Locate predeclared procedure of use. It is described in the section "Miscellaneous Topics." This can be used to load registers with the Amdahl/470 address of an Algol W variable or expression result. It can also be used with arrays to build special or nonstandard S-type parameter lists which the Call predeclared procedure cannot handle directly.

Example:

```
integer Aofreg;
 .
 .
R0 := 3, R1 := 8192;
Rcall("GETSPACE");
Aofreg := R1;
 .
 .
R0 := 0; R1 := Aofreg;
Rcall("FREESPAC");
```

This example shows a section of an Algol W program, running under MTS, which is performing its own storage management by calling the system subroutines GETSPACE, to acquire storage, and later FREESPAC to release it again. Both of these subroutines have calling sequences which use and return values in machine general registers. They are described in MTS Volume 3, System Subroutine Descriptions.

Normally Algol W will perform all necessary management of storage regions needed by a program. The above example is given only to demonstrate Rcall. The only situation where a program might need to perform its own storage management would be if extremely large arrays were required. Note that, if this were the case, the predeclared procedures Store and Fetch would be needed to interface between storage so acquired and normal Algol W variables. They are described in the section "Miscellaneous Topics."

## The FORTRAN External Reference

This mechanism was the original one supplied by the previous version of Algol W to call S-type subroutines. Its use is no longer recommended - see instead the previous subsection on the Call predeclared procedure. The 'fortran' external reference is documented here for completeness so that a coding of older Algol W source programs may be understood.

In order to call a FORTRAN subroutine using this mechanism it is necessary to declare a procedure heading describing the parameters of the subroutine, followed by an external reference. An external reference for a FORTRAN subroutine consists of the reserved word 'fortran' followed by a string constant containing the name of the subroutine to be called. Such a phrase stands as a procedure body in a procedure declaration and establishes the connection between the Algol W program or procedure and the subroutine.

The procedure so declared is then called in the same way as for an externally defined Algol W procedure. However there are some restrictions in the way parameters may be passed between Algol W and FORTRAN. The type correspondence between Algol W and FORTRAN has been given previously in this section.

The legal formal parameters and their meanings in an Algol W procedure linked to a FORTRAN subroutine are as follows:

Call by name

> The passing of the corresponding argument is treated as a call by reference. If the argument is a variable, the address of that variable is computed and passed. If the argument is an expression, the value of the expression is determined (only once), assigned to an internal local variable and the address of that variable is passed.

Call by value, call by result, call by value result

> The passing of the corresponding argument is treated as call by value, call by result or call by value result, respectively.

Array call by name

> Array element addresses are passed with no subscript checking. In other words, even if the element lies outside the bounds of the Algol W array, its address is passed. Arrays with a lower bound (or bounds) of 1 can match FORTRAN parameters exactly. Partial arrays should not be used as arguments.

Note that a procedure is not a legal formal parameter in an Algol W procedure declaration corresponding to a FORTRAN subroutine.

## PARAMETER TYPE CORRESPONDENCE FOR EXTERNAL SUBROUTINES

Algol W has nine simple types of variable; the correspondence between these types and the representation of data on System/370 type machines is as follows:

| Algol W Simple Type | Machine Representation | |
|---|---|---|
| integer | F | Fixed-point; 4 byte 2's complement number |
| real | E | Short precision floating-point; 4 bytes |
| long real | D | Long precision floating-point; 8 bytes |
| complex | 2E | Two adjacent type E's; 2 times 4 bytes |
| long complex | 2D | Two adjacent type D's; 2 times 8 bytes |
| logical | X | Byte; X'00' = false, X'01' = true; |
| string(n) | CLn | n adjacent bytes; one byte/character |
| bits | F | Fixed-point; 4 bytes = 32 bits |
| reference | A | Address of a data structure; 4 bytes |

Notes:

(1) Integer, real, complex, bits, and reference variables are said to be fullword aligned; that is each data item starts at a machine address which is exactly divisible by four.

(2) Long real and long complex variables are said to be doubleword aligned; that is each data item starts at a machine address which is exactly divisible by eight.

(3) Logical and string variables are nonaligned; they may start on any machine address boundary.

(4) Complex and long complex variables occupy two adjacent storage locations of the relevant type. The first location contains the real or long real part and the second the imaginary or long imaginary. There are no complex operations defined on System/370 type hardware; Algol W simulates such operations by routines operating on the real and imaginary components as ordinary floating-point numbers.

(5) String character encodings are given in Appendix B.

(6) Reference variables hold the address of the data structure to which they are pointers. However some knowledge of Algol W implementation beyond that given in this manual is necessary to predict the location of a designated field within the record.

Frequently external subroutines called from an Algol W program will have been written in FORTRAN. FORTRAN supports a range of data types; the correspondence between IBM FORTRAN IV and Algol W simple types is as follows:

| FORTRAN IV Data Type | Algol W Simple Type |
|---|---|
| INTEGER | integer or bits |
| INTEGER*2 | -- see note (1) |
| INTEGER*4 | integer or bits |
| REAL | real |
| REAL*4 | real |
| REAL*8 | long real |
| COMPLEX | complex |
| COMPLEX*8 | complex |
| COMPLEX*16 | long complex |
| DOUBLE PRECISION | -- see note (2) |
| LOGICAL | integer or bits |
| LOGICAL*1 | logical |
| LOGICAL*4 | integer or bits |
| -- see note (3) | string(n) |

Notes:

(1) INTEGER*2 in FORTRAN specifies the use of a 16-bit halfword integer (System/370 type H data). Algol W does not support such 'short' integers. Single (that is scalar) parameters of this type can be fetched and retrieved using a combination of Algol W integers and the predeclared functions Halfword and Fullword. These are described in the section "Miscellaneous Topics." Where arrays of halfwords have to be transmitted or received, considerably more effort would be required to build the required vectors. The Move predeclared procedure may be of use; it is also described in the section "Miscellaneous Topics."

(2) A FORTRAN DOUBLE PRECISION statement will make a REAL into a REAL*8 which is Algol W long real and a COMPLEX into a COMPLEX*16 which is Algol W long complex.

(3) Algol W strings have no directly equivalent data type in FORTRAN. A dimensioned LOGICAL*1 variable may be used; but in other circumstances a FORTRAN programmer might decide that an integer vector was more appropriate with four characters packed into each element. Each instance must be considered separately.

# MISCELLANEOUS TOPICS

This section gives details of miscellaneous facilities provided by Algol W which cannot easily be categorized under any of the other sections in this manual.

## PREDECLARED STATE VARIABLES

The following predeclared variables give machine constants or status values. They may be considered to be declared in a block global to the entire Algol W program and may therefore be redeclared within a program. If this is done they will of course lose their initial meanings and values.

Note that as these values are held in variables they may be changed by reassignment. However this is bad practice since it may well confuse someone who subsequently has to read the program text.

A full list of all predeclared variables is given in Appendix G. Those not listed here have been described elsewhere in the manual.

logical Canreply

This variable is initialized to 'true' if the executing program is being run at a conversational terminal, or 'false' if the program is being run in batch. It may therefore be used in tests to decide whether a user prompt is in order. For example:

```
Putcard(Error, "0Invalid command to GLURP Processor");
if Canreply then
begin
   Putcard(Error, "&Enter new command..");
   Getcard(User, Prompt_Input)
end else
begin
   Putcard(Error, "0Error recovery fails...");
   Putcard(Error, "0Batch program termination...");
   Stop(null)
end Prompt_If_Blocks;
```

This partial program tests the value of Canreply and performs a different action depending on whether the user is at a conversational terminal or has previously submitted the program to the MTS batch stream. The Stop predeclared procedure is described later in this section.

real Epsilon

The initial value of 9.536743'-07 (hexadecimal 3BFFFFFF) in this variable is the largest positive real number e provided by the implementation such that: 1 + e = 1. See Appendix J for details of the limitation of representation of Algol W real values on System/370 type machines.

long real Longepsilon

The initial value of 2.22044604925031'-15L (hexadecimal 33FFFFFFFFFFFFFF) in this variable is the largest positive long real number e provided by the implementation such that: 1L + e = 1L. See Appendix J for details of the limitation of representation of Algol W long real values on System/370 type machines.

integer Maxinteger

The initial value of 2147483647 (hexadecimal 7FFFFFFF) in this variable is the maximum positive integer which is allowed by the implementation. See Appendix J for a discussion of the representation of Algol W integer values on System/370 type machines.

long real Maxreal

The initial value of 7.23700557733225'+75L (hexadecimal 7FFFFFFFFFFFFFFF) in this variable is the largest positive long real number allowed by the implementation. See Appendix J for a discussion of the representation of floating point values on System/370 type machines.

long real Pi

The initial value = 3.14159265358979L (hexadecimal 413243F6A888-5A31) in this variable is the best approximation to the familiar mathematical constant which is the ratio of the circumference to diameter of a circle available on System/370 type machines.

string(256) Sysparm

This variable is provided to allow a single record of input data to be supplied to an Algol W program when it is run. See the description under "Run-Time Parameters" in the section "Algol W Programmer's Guide." See also the description of the DATAPARM parameter in the same section.

(1) If no run-time parameter is specified Sysparm will contain 256 blanks.

(2) If RUNPARM=DATAPARM is not given but a run-time parameter string contains a string enclosed by quote (") or prime (') delimiters then the contents of the last such string specified are placed in Sysparm.

(3) If RUNPARM=DATAPARM is given at compile time then the entire unedited contents of the run-time parameter string are placed in Sysparm.

The following program takes advantage of this:

```
begin
   integer Datavalue;
   if Sysparm = " " then
   begin
      Putcard(Error, "0Enter data value");
      Get(User, Datavalue)
   end else
      Getstring(Sysparm, null, Datavalue);
    .
    .
end.
```

The program checks to see if a value has been supplied in the run time parameter string: if so it decodes it using predeclared procedure Getstring. If no run parameter has been given the program requests one.

## CLOCK FUNCTIONS

The Algol W environment includes a clock which measures the elapsed time since the beginning of program execution, the time of day, and the date. The precision of the clock is determined by the argument given by the user, and ranges from 1/60 second to 1/38400 second. Two predeclared functions are provided for reading the clock, Time and Date.

### Time

Time returns an integer result. The general form of the function call is:

Time(<integer-expression>)

The result returned depends on the value of the integer expression as follows:

Time(-2)

Returns the elapsed time since the program started execution in units of 1/60 seconds. Note that both this argument and Time(-1) provide a real time clock.

Time(-1)

Returns the time since midnight in units of 1/60 seconds. This provides a basic time of day clock. For many applications the time of day string function provided by Date (see next section) may be of more interest.

The remaining arguments to Time all provide measures of the execution time since the program started running. This is not real time; it is the time the computer's central processor unit has spent servicing the task calling the Time function. They therefore provide various measures of the work done by the executing program. The term CPU time is used to encompass these values.

Problem state CPU time is the time the central processor spends executing the Algol W program and library code. Supervisor state CPU time is time spent in system service routines such as those providing physical input/output operations. Total CPU time is the sum of these two times. When comparing the CPU time requirements for numerical, combinatoric or related algorithms the problem state times should be compared. Supervisor, and therefore total, CPU time is very dependent on total system loading.

Time(0)

Returns the total CPU time for this program in units of 1/100 minutes.

Time(1)

Returns the total CPU time for this program in units of 1/60 seconds.

Time(2)

Returns the total CPU time for this program in units of 1/38400 seconds.

Time(3)

Returns the problem state CPU time for this program in units of 1/38400 seconds.

Time(4)

Returns the supervisor state CPU time for this program in units of 1/38400 seconds.

The result for any other argument is undefined.

Date

Date provides a 24-character string encoding of the time of day and date. The general form of the function call is:

Date(<integer-expression>)

Two formats are provided:

Date(0)

This argument returns a string value which is 24 characters long. This value is of fixed format so that relevant items from the time of day display can be easily extracted by use of substring designators, the Decode predeclared function (see the section "Strings") or the Getstring predeclared procedure (see the section "Stream Directed Input and Output"). The basic format of the returned string is:

␣<time>␣<date>␣<weekday>␣<yearday>

where:

␣ is a single blank;

<time> is an eight character representation of the time on the 24 hour clock as: "hh:mm:ss";

<date> is an eight character representation of the date as "mm:dd:yy" (note the American form where the month is given first);

<weekday> is a single character number giving the day of the week (running from 1 for Sunday to 7 for Saturday) and

<yearday> is a three character number giving the day of the year (January first is counted as day one and leading zeros are not suppressed).

For example:

" 15:08:40 03-25-80 3 085"

shows a time in the afternoon of Tuesday (day 3), the 25th of March 1980. This is the 85th day of the year.

Date(1)

This argument provides a time and date suitable for direct printing in output. The format is:

␣<time>␣<half>␣␣<weekday>␣<day>␣<month>␣<year>

where:

<time> is a five character time of day on the twelve hour clock as "hh:mm";

<half> is "am" or "pm" according to the time of day;

<weekday> is is a three character day of the week from the set: "Sun" | "Mon" | "Tue" | "Wed" | "Thu" | "Fri" | "Sat" ;

<day> is a two character number giving the day of the month (leading zero if any is suppressed);

<month> is a three character day of the year from the set: "Jan" | "Feb" | "Mar" | "Apr" | "May" | "Jun" | "Jul" | "Aug" | "Sep" | "Oct" | "Nov" | "Dec" ;

<year> is a two character number giving the year.

For example:

" 3:08 pm Tue 25 Mar 80"

The result for any other argument to date is undefined.

## EXCEPTIONAL CONDITIONS

The facilities described below are provided in Algol W to allow detection and control of certain exceptional conditions arising in the evaluation of arithmetic expressions and predeclared functions.

The following predeclared record class is provided:

```
record Exception(
   logical Xcpnoted;
   integer Xcplimit,
           Xcpaction;
   logical Xcpmark;
   string(64) Xcpmsg );
```

Associated with each exceptional condition which can be processed is a predeclared reference variable to which record occurrences of class Exception can be assigned. Fields of such record occurrences control the processing of exceptions. The implicit declaration is:

```
reference(Exception) Endfile, Ovfl, Unfl, Divzero,
                     Intdivzero, Intovfl, Function;
```

The association between exceptional conditions and reference variables is as follows:

| Reference Variable | Exceptional Condition |
|---|---|
| Endfile | end-of-file detected on input by the predeclared procedures Read, Readon, Readcard, Get or Geton |
| Ovfl | real, long real, complex or long complex exponent overflow |
| Unfl | real, long real, complex or long complex exponent overflow |
| Divzero | real, long real, complex or long complex division by zero |
| Intovfl | integer overflow |
| Intdivzero | integer division by zero |
| Function | all domain, significance, and singularity errors in the evaluation of mathematical predeclared functions |

When one of the conditions listed above occurs one of two actions is followed, depending on whether or not the corresponding reference variable is set to the reference constant 'null'.

If the value is 'null' the exceptional condition is ignored and execution of the Algol W program continues. In such conditions the predeclared functions return a default value - see "Predeclared Function Errors and Default Values" later in this section. All other conditions return the result provided by the underlying computer hardware - see "Table of Results for Exceptional Conditions" later in this section.

Note that, excluding the reference variable Unfl, the only way for a predeclared reference variable to be 'null' is if it is specifically set by the user. Initially Algol W sets all of the above references to point to a special system record occurrence. This is not available to the user by name. The statement:

```
Unfl := Ovfl;
```

would set Unfl so that it pointed to this special reference, provided Ovfl had not been reassigned prior to this.

Exception - Field Values

If the value of a given reference variable is not 'null' and the corresponding condition occurs, the fields of the given record occurrence of class Exception are checked. Depending on their values various actions are taken. The following descriptions explain the meanings of each field of the record class Exception:

Xcpnoted

This is a logical variable used to note the occurrence of the exceptional condition. If the corresponding reference variable is not null, this variable is always set to 'true' when the corresponding exceptional condition occurs. It is sensible for the user to initialize this variable to 'false' so that the program can test for an exception of the given kind.

Xcplimit

This is an integer variable used to set a limit on the number of times the user wishes to allow the given exception to occur before execution is ended. When Xcplimit is reached a fatal error message is returned. Run error messages are listed by number in Appendix C.

Xcpaction

This is an integer variable whose value determines the resulting value of the expression (that is the result of the arithmetic evaluation, predeclared function call or input operation) in which the exception occurred. If execution is allowed to continue (see Xcplimit above), the new value of the expression is set according to the following values of Xcpaction:

| Value | Action |
|---|---|
| 1 | The resulting value of the variable follows the "Adjustment Table" described later in this section. |
| 2 | The result is long real zero. |
| ¬= 1 or 2 | The result is the Algol W default shown in the first column of the "Table of Results for Exceptional Conditions" later in this section. |

Note that Xcpaction is only relevant for reference variables which correspond to "Special Conditions" (see below). If the current reference variable is Intovfl or Intdivzero, the resulting value of the expression in which the exception occurred is the Algol W default.

Xcpmark

This is a logical variable to be set by the user. If the value is 'true' an error message is printed in the event of an exception of the given kind. The error message may include the text of the variable Xcpmsg - see below. Note that an error message is always printed when execution is ended by an exception. This means that the value of Xcpmark is only relevant if Xcplimit has not yet been reached.

Xcpmsg

This is a string variable of length 64. If an error or exception message is printed, the contents of Xcpmsg will appear in the text of the message if Xcpmark has been assigned and if Xcpmsg is nonblank.

## Special Conditions and Adjustment Table

There are two sets of exceptional conditions:

(1) Special conditions. This includes all exceptional conditions except those processed using Intovfl and Intdivzero.

(2) Integer overflow (Intovfl) and integer division by zero (Intdivzero).

If the field values of the record occurrence of class Exception specify that execution is to continue, action then depends on which set the current reference belongs to.

If the reference variable is either Intovfl or Intdivzero, the result is the Algol W default given in the first column of the "Table of Results for Exceptional Conditions" later in this section.

If the reference variable is one of the special conditions, the result is determined by the value of Xcpaction. The following table gives the values of the resultant expression when Xcpaction is set to 1.

Adjustment Table

| Special Condition | Adjustment of Result |
|---|---|
| Ovfl, Divzero | if Algol W default < 0<br>then -Maxreal else Maxreal |
| Unfl | long real zero |
| Function | the value of the long real<br>predeclared variable Fn_Value |
| Endfile | according to type:<br>numerical 0 ; string " " ;<br>logical false ; bits #0 |

The special condition, Function, occurs when arguments are out of domain for the predeclared function which caused the exception. A full list of predeclared functions with their domains of definition is given in Appendix F.

Table of Results for Exceptional Conditions

| Condition | Algol W default: Xcpaction ¬= 1 or 2 | Xcpaction = 1 | Xcpaction = 2 | Hardware default: reference = null |
|---|---|---|---|---|
| Endfile | 0 | 0 | 0 | 0 |
| Ovfl | exponent 128 too small | ±Maxreal | 0 | exponent 128 too small |
| Unfl | exponent 128 too large | 0 | 0 | 0 |
| Divzero | dividend | ±Maxreal | 0 | dividend |
| Intovfl | true result ±2**32 | true result ±2**32 | true result ±2**32 | true result ±2**32 |
| Intdivzero | dividend | dividend | dividend | dividend |
| Function | see the next section | Fn_Value | 0 | see the next section |

Notes:

(1) When an end-of-file condition occurs when attempting to read a string, a string of blanks is returned; for a logical, false is returned; for bits #0.

(2) Default values for the Function reference are discussed in the Section "Predeclared Function Errors and Default Values" which follows this one.

Note that the results of the nonspecial conditions Intovfl and Intdivzero are not affected at all by either the Xcpaction value or whether or not the reference variable is set to null. The only reason for assigning values to fields of Exception for these variables is to have a special Xcpmsg value printed and to be able to continue execution past the first occurrence of the condition. The same is true of Endfile.

Examples:

```
Ovfl := Exception(false, 10, 1,
             true, "Overflow circumvented");
```

The field values and their effects are:

```
Xcpnoted   false   becomes true if overflow occurs
Xcplimit   10      allows up to 10 overflows
Xcpaction  1       replaces result with ±Maxreal
Xcpmark    true    prints Xcpmsg each time
Xcpmsg     "Overflow circumvented"
```

Another example:

```
Intdivzero := Exception(false, 50, 2, false, );
```

The field values and their effects are:

```
Xcpnoted   false   becomes true if overflow occurs
Xcplimit   50      allows up to 50 integer division by zeros
Xcpaction  2       irrelevant; always replaced by dividend
Xcpmark    false   no message printed until final termination
Xcpmsg     not assigned
```

## Predeclared Function Errors and Default Values

The floating point predeclared functions are not always valid for all values of their arguments. When invalid arguments are passed, a predeclared function exceptional condition is recognized and Algol W processes this under the control of the predeclared reference variable Function.

Predeclared functions whose exceptions are processed under the control of the reference variable Function are described in the section "Arithmetic Expressions and Assignment Statements" in the subsections:

"Roots and Powers Functions"
"Trigonometric Functions"
"Inverse Trigonometric Functions"
"Hyperbolic Functions"
"Special Functions"
"Complex Functions"

A list of predeclared functions giving domains of definition and singularities, if any, is given in Appendix F. Exceptional conditions arising from the use of these functions are described in this section.

If Xcpaction(Function) is 2, long real zero is returned. If Xcpaction(Function) is 1, the value of the predeclared long real variable Fn_Value is returned. For a complex function the imaginary part will be zero.

If the reference variable Function is 'null' or Xcpaction(Function) is zero, a default value is returned. This default value depends on the function called and the argument passed. The default values for the floating point predeclared functions are given with the description of the function definitions below.

Square Root

The real valued functions Sqrt and Longsqrt are not defined for negative arguments since the square root of a negative number is pure imaginary. If x < 0 then:

```
x**0.5 = i•|x|**0.5
```

The default function value computed in this case is the square root of the absolute value of the argument.

The complex square root functions Cxsqrt and Longcxsqrt are defined for all values of their arguments.

Common and Natural Logarithms

The real valued logarithmic functions Log, Ln, Longlog and Longln are not defined for negative arguments since the logarithm of a negative number is complex. If x < 0 then:

```
ln(x) = ln(|x|) - i•Pi
```

The default function value is the logarithm of the absolute value of the argument.

All of the logarithmic predeclared functions are undefined for an argument of zero. This is a singularity in the definition of the

logarithm function. The default returned value is minus machine infinity, which is approximately -7.2'+75.

Exponential

The real valued functions Exp and Longexp can be properly defined only within the interval from -180.2182 to +174.67308 because of the range restrictions imposed by the representation of floating point numbers on System/370 type machines - see Appendix J. The two extremes for argument values quoted above are the natural logarithms of minus and plus machine infinity.

If the argument exceeds the upper limit, the default function value is machine infinity.

If the argument value is less than the lower limit, the default value is zero. However, this situation is only regarded as an error if the floating point underflow exception is being trapped; that is the predeclared reference value Unfl has been assigned a value other than the default of 'null'.

It should be noted that the domain of definition for the exponential functions is slightly less than the range of the corresponding natural logarithm functions. Hence the expressions Exp(Ln(X)) and Longexp(Longln(X)) are not computable for values of X extremely close to the ends of the machine range.

The complex valued functions Cxexp and Longcxexp have an analogous restriction on the real part of the complex argument and an additional restriction on the imaginary part. The function definitions are:

$$cxexp(x+i\cdot y) = exp(x)\cdot[cos(y)+i\cdot sin(y)]$$

The domain restrictions on the imaginary part are therefore those of the Sin, Longsin, Cos and Longcos functions.

Trigonometric Functions

The domain restrictions for the real valued trigonometric functions Cos, Sin, Tan, Cot, Longcos, Longsin, Longtan and Longcot are imposed to maintain accuracy. These functions are computed by reducing the argument to the interval from -Pi/4 to +Pi/4 by taking advantage of the periodicity of the function values. For very large arguments this reduction yields so few significant digits in the reduced argument that meaningful computation of the function value is impossible.

The single precision functions require an argument whose absolute value is less than 823549.563 (or approximately 2**18•Pi).

The long precision functions require an argument whose absolute value is less than 3.537118706008'+15 (or approximately 2**50•Pi).

The default function value for arguments outside this range is zero for all real valued trigonometric functions.

In addition, the tangent and cotangent functions will object if the argument is too close to one of their singularities to maintain accuracy or if the function value lies outside the machine range. In these situations, the default function value is machine infinity with the sign of the argument.

The complex sine and cosine functions Cxcos, Cxsin, Longcxcos and Longcxsin may be defined as:

$$cxsin(x+i \cdot y) = sin(x) \cdot cosh(y) + i \cdot cos(x) \cdot sinh(y)$$

$$cxcos(x+i \cdot y) = cos(x) \cdot cosh(y) + i \cdot sin(x) \cdot sinh(y)$$

These formulae illustrate why a trigonometric type of domain restriction is applied to x and an exponential type to y. The default function value is derived from the default values supplied by the appropriate sine, cosine and exponential functions, where cosh(y) and sinh(y) become machine infinity divided by two when |y| is too large.

## Inverse Trigonometric Functions

The domain of the inverse sine and cosine functions Arccos, Arcsin, Longarccos and Longarcsin is the range of the sine and cosine functions, that is from -1 to 1. Outside this interval the default function value is zero.

The inverse tangent functions Arctan and Longarctan are defined for all values of their arguments.

Note that these functions return the principal values of functions which are, in fact, many valued.

## Hyperbolic Functions

The value of the hyperbolic sine and cosine of x exceed the range of the machine when |x| approaches the logarithm of machine infinity. The functions Cosh, Sinh, Longcosh and Longsinh are defined for arguments whose absolute values are less than 175.3662.

The default function value is machine infinity with the appropriate sign.

The hyperbolic tangent functions Tanh and Longtanh are defined for all values of their arguments.

## Special Functions

The gamma function is supplied by the Gamma and Longgamma predeclared functions and the logarithmic gamma function by Lngamma and

Longlngamma. Like the exponential functions, these exceed the machine range for arguments outside their domains of definition.

Maxreal is the default value for both sets of functions.

The error functions Erf and Long erfc and the complementary error functions Erfc and Longerfc are defined for all values of their arguments.

## ADDITIONAL IOCONTROL OPTIONS

The predeclared procedure Iocontrol was introduced in the section "Basic Input and Output." Additional keywords are available to provide extended control over certain system actions. As indicated in the previous description of Iocontrol, integer arguments are available corresponding to each of the keywords. However use of keywords is recommended as it is less prone to error. These integers are in parentheses following the keywords listed below.

### Timing Information

When an Algol W program finishes execution, timing information about the run is normally printed. By default Algol W will print the total CPU time for the run unless the NOTIMES keyword is given - see below. The keywords described below allow other aspects of the run timing to be displayed on termination or, alternatively, the timing information may be suppressed altogether.

All of the keywords described in this particular subsection are also available as run-time parameters. See the section "Algol W Programmer's Guide."

TOTALCPU (101)

This keyword causes the total CPU time to be printed. This is the default. This total time is the sum of the times which the user's task spent in both problem and supervisor state during execution of the Algol W program at run-time. This then is the time which the computer spent in running the user's program. It is always much less than the elapsed real time for the run because the computer is servicing many users simultaneously. The difference between problem and supervisor state is discussed below.

PROBLEMCPU (102)

This keyword causes the time spent in problem state during the program's execution to be printed. All normal computation is done

in problem state. When comparing program timings in an attempt to assess efficiency of various algorithms, the problem state time should be inspected. This time is least likely to be affected by variations in the total machine loading.

SUPERCPU (103)

This keyword causes the time spent in supervisor state during program execution to be printed. If examined by itself the supervisor time alone can give a misleading impression of program timing, so this keyword also forces printing of the problem state CPU time. The machine runs in supervisor state when certain privileged operations must be performed. Typically this occurs when program action causes direct input/output to the devices attached to the computer. Such time is charged to the user as supervisor state time but it should be born in mind that the time to access a particular input/output device is very dependent on the competition for the various devices from all the machine's various tasks. This time therefore rises steeply at times of peak machine load and should not be used as more than a general guide to program efficiency. Programs which do a great deal of input/output are said to be input/output bound and can be expected to spend a relatively large percentage of their total CPU time in supervisor state.

ELAPSED (104)

This keyword prints the elapsed real time for the program's execution, that is it prints the difference between the time of day at program termination and that at the start of program execution. It is printed in minutes and seconds.

NOTIMES (105)

This keyword suppresses the printing of all timing information. If the total CPU time is not required but one of the other options is, then two keywords can be given to achieve this effect:

```
Iocontrol("NOTIMES,ELAPSED");
```

This would cause only the elapsed time for the run to be printed.

ALLTIMES (106)

This keyword combines the effect of TOTALCPU, PROBLEMCPU, SUPERCPU and ELAPSED. It therefore causes Algol W to print a complete timing information breakdown on termination.

External and Library Interruptions

When a program interruption occurs during an Algol W program execution, control is passed to the Algol W error processor. If the program interruption occurs during execution of the user's portion of an Algol W program or as a result of erroneous data supplied by a user program to an Algol W library routine, then the exception is handled according to the values of the predeclared references to the predeclared Exception record. This can be suppressed by specifying the NOTRAPS run-time option.

If, on the other hand, the program interrupt occurs as a result of either a genuine error in the coding of an Algol W library routine or while an external non-Algol W routine is being executed, then the behavior of the error processor is different. A fatal error message is written out indicating whether the interruption occurred in a library routine or an external routine.

If the error message indicates a library routine is at fault, please notify a member of the Computing Center Staff. If an external routine is the cause of the trouble, then either its coding or the parameters supplied to it are in error and the problem should be diagnosed on this basis.

In either case the format of the subsequent error message and the manner of program termination may be controlled by the following Iocontrol keywords:

DISPLAY (201)

> This keyword causes a dump of the value of the program status word, the sixteen general registers and the four floating point registers to be printed on the ERROR stream. (In MTS this is SERCOM.) This is the default action.

NODISPLAY (202)

> This keyword completely suppresses the program status word and register dump.

PSW (203)

> This keyword causes the value of the program status word to be printed.

GRS (204)

> This keyword causes the values of the 16 general registers to be printed.

FRS(205)

This keyword causes the values of the four floating point registers to be printed.

If the program status word alone is required then the following statement should be issued:

```
Iocontrol("NODISPLAY,PSW");
```

The remaining two keywords control the action which Algol W will take when processing of the error condition is complete.

SYSPGNT (206)

This keyword specifies that control is to be passed to the operating system with the registers and program status word having the values at the point of the interrupt. In MTS issuing this keyword can be used as a method of forcing program interrupts in external non-Algol W routines to pass control to the Symbolic Debugging System. The cause of the error can then be investigated further by using SDS commands. Note that this keyword has no effect on the processing of program interrupts which occur within Algol W coding. Such exceptions are always handled by Algol W in the usual fashion.

ALWPGNT (207)

This keyword is the converse of SYSPGNT and is the default action. Program interruptions in external and library routines cause a normal program termination. If the program execution had been initiated in compile, load and go mode then control is returned to the Algol W system, allowing reexecution of the program, system termination or compilation of a corrected or a new program as desired.

## Control of Getstring Action

The Getstring predeclared procedure allows a user program to perform input data conversion in such a way that it can retain control if a fatal conversion error occurs. It also enables simple free format command scanners to be written easily in Algol W.

In order to achieve these aims the action of Getstring may be influenced by the following Iocontrol keywords:

GSFIELDED (301)

This keyword tells the Algol W system that data conversion errors should cause a fatal error condition to be recognized and that

execution of the program should be terminated with the relevant error message. This action is the default. Note that a free format Getstring operation, which fails to find characters other than break characters between the start of the scan and the end of the string, will be recognized as a fatal error. Getstring cannot fetch a new input string in the same way that the ordinary input predeclared procedures can fetch a new physical input record.

GSRETURNS (302)

This keyword tells Algol W that in the event of a failure of the Getstring routine, control is to be returned to the user program. On return the predeclared integer variable Syscode will contain a value indicating the success or failure of the conversion operation.

| Syscode Value | Meaning |
|---|---|
| 0 | Conversion was successful. |
| 4 | While scanning for an item the end of the string was encountered, that is no data item was present. |
| >4 | A data item was found but it could not be converted to the internal form implied by the simple type of the receiving variable. |

The remaining two keywords allow a subsequent Getstring operation to continue scanning from the point in the string at which a previous Getstring operation terminated. Note that the information held between Getstring calls is only the tab position down the string at the end of the first call. The onus is on the user to code a program in such a way that further Getstring calls specify the same string to be scanned.

GSCONTINUE (303)

GSCONTINUE specifies that the tab position at the end of a Getstring call is to be remembered by the Algol W system and used as the starting position in a subsequent Getstring call. This allows coding of a command scanner in such a way that it may pick up parameters from the string individually and deal with them between Getstring calls.

GSORIGIN (304)

GSORIGIN specifies that any remembered tab position which might be used for the next Getstring call is to be forgotten. Any subsequent Getstring calls will start their scans from the beginning of the supplied strings.

An example of the use of these keywords is given in the Section "A Simple Command Scanner" later in this section.

## Modification of the String Recognition Algorithm

The algorithm by which strings are recognized as data items in an input stream has been described in the subsection "Strings" in the section "Basic Input and Output." While this algorithm is sufficient for normal data input there are many occasions when modifications to it would be useful. For instance the delimiting action of quotes and primes in the input data might not be desirable when coding a command scanner.

While the coding of more general scanning algorithms, such as those used in lexical analysis, must be left to the user, the following Iocontrol keywords provide the ability to implement scanners for the more common types of arguments in command driven programs.

RESETSCAN (401)

RESETSCAN specifies that the string recognition algorithm is to revert to the default action, that is it undoes the action of any of the keywords described in this section which change the default action.

NOQUOTES (402)

This keyword specifies that the quote (") is not to be recognized as a string delimiter in subsequent string input operations.

In the section of program:

```
string(32) S, T;
 .
 .
Iocontrol("NOQUOTES");
Read(S, T);
```

a data card containing:

```
ABC "DEF GHI"
```

would cause S to contain ABC and T to contain "DEF .

QUOTES (403)

This keyword undoes the effect of NOQUOTES. In subsequent string input the quote is recognized as a string delimiter.

NOPRIMES (404)

This keyword specifies that the prime (') is not to be recognized as a string delimiter in subsequent string input operations.

In the section of program:

```
string(32) S, T;
 .
 .
Iocontrol("NOPRIMES");
Read(S, T);
```

a data card containing:

```
' IS A PRIME
```

would cause S to contain ' and T to contain IS.

PRIMES (405)

This keyword undoes the effect of NOPRIMES. In subsequent string input the prime is recognized as a string delimiter.

BRACKETS (406)

The BRACKETS keyword specifies that a group of characters enclosed in parentheses is to be recognized as a string, including the parenthesis characters. Nesting of parentheses is recognized. The outer parentheses are said to be zero level and a string which starts with a left parenthesis is terminated by the next zero level right parenthesis followed by a break character or the end of the string.

With the section of program:

```
string(32) S, T;
 .
 .
Iocontrol("BRACKETS");
Read(S, T);
```

the data card containing:

```
(A, B, C),(FMT=VB(3200,80), SL),XYZ
```

would cause S to contain (A, B, C) and T (FMT=FB(3200,80), SL)

NOBRACKETS (407)

NOBRACKETS specifies that groups within parentheses are not to be specially treated. It undoes the effect of both the previous keyword, BRACKETS, and the next keyword, DELBRACKETS.

DELBRACKETS (408)

DELBRACKETS causes the same string recognition of parentheses as BRACKETS but when the string conversion takes place the outer, zero level, parentheses are stripped from the string. If the string so recognized is not completely enclosed in zero level parentheses then no stripping of parentheses takes place.

If DELBRACKETS had been given instead of BRACKETS in the previous example, then the string S would contain A, B, C and string T FMT=FB(3200,80), SL. However if the data item had been:

```
(A, B, C),FMT=FB(3200,80),XYZ
```

then string S would be the same but string T would contain FMT=FB(3200,80) that is the parentheses would not be removed because they do not enclose the whole of the string so recognized.

These keywords are particularly useful when used to construct command scanners by the use of the Getstring predeclared procedure. An example of the use of such a scanning algorithm is given in the next Section "A Simple Command Scanner."

## A Simple Command Scanner

The following small program demonstrates some of the Iocontrol features described in previous sections. Note that the program is compilable but not complete; the procedures Add_Entry, Delete_Entry and Print_Entry are present in embryo form only.

```
begin
   record List(string(256) Item; reference(List) Next);

   reference(List) procedure Scan_Input_Text(
      string(256) value Input_Data);
   begin
      comment Break down a command into its constituents;
      reference(List) First, Last;
      string(256) Chars;
      integer Save_Code;

      comment
         Initialize linked list to null and use Iocontrol
         keywords to change the string recognition algorithm
         and prevent Getstring from invoking the error processor;

      First := null;
      Iocontrol("GSRETURNS,NOQUOTES,NOPRIMES");
```

```
      comment
         Use Getstring in a 'while' loop to scan the string
         building a linked list of each command element;

      while
      begin
         Getstring(Input_Data, null, Chars);
         Save_Code := Syscode;
         Iocontrol("GSCONTINUE");
         Save_Code = 0
      end do
      if First = null then
         First := Last := List(Chars, null)
      else
         Last := Next(Last) := List(Chars, null);

      comment Reset the defaults and return the linked list;
      Iocontrol("RESETSCAN,GSORIGIN");
      First
   end Scan_Input_Text;

   procedure Process_Command(reference(List) value Cmd);
   if Cmd ¬= null then
   begin
      logical Found;
      integer Cmd_Number;
      string(32) Cmd_Name;

      comment
         Extract command name from Cmd record and
         skip reference past command name ready for
         individual command processors.  The command
         name is translated to upper case in case it
         has been entered in mixed case from the terminal ;

      Cmd_Name := Item(Cmd)(0//32);  Cmd := Next(Cmd);
      Translate(Cmd_Name, Uppercase);

      comment
         Initialize 'while' loop variables.  The loop
         increments the command number and uses a case
         statement to check for possible command strings.
         Note that this serial search is efficient for a
         small number of command names only.  Binary
         search or computed key (hash table) methods are
         more suitable for generalized keyword identification ;

      Cmd_Number := 0;  Found := false;
      while ¬ Found do
      begin
         Cmd_Number := Cmd_Number + 1;
         case Cmd_Number of
         begin
```

```
            if Cmd_Name = "ADD" then
            begin  % Execute ADD command %
               Found := true;  Add_Entry(Cmd)
            end Add_Check_Block;

            if Cmd_Name = "DELETE" then
            begin  % Execute DELETE command %
               Found := true;  Delete_Entry(Cmd)
            end Delete_Check_Block;

            if Cmd_Name = "PRINT" then
            begin  % Execute PRINT command %
               Found := true;  Print_entry(cmd)
            end Print_Check_Block;

            begin  % Command unidentifiable %
               Found := true;
               Put(Error, "/X,H"",A,T,H"",X,A",
                   Cmd_Name, "is invalid - ignored")
            end Invalid_Command_Block

         end Case_Block
      end
   end Process_Command;

   procedure Add_Entry(reference(List) value Cmd_Parms);
   begin
      Putcard(Error, "      *** Add Entry ***")
   end add_entry;

   procedure Delete_Entry(reference(List) value Cmd_Parms);
   begin
      Putcard(Error, "      *** Delete Entry ***")
   end delete_entry;

   procedure Print_Entry(reference(List) value Cmd_Parms);
   begin
      Putcard(Error, "      *** Print Entry ***")
   end print_entry;

   comment Main program;
   reference(List) Command;
   string(256) Line;

   comment  acquire catalog file;
   Assign("CATALOG", "CATFILE");

   comment
      Now loop to read in and process commands.
      Note the use of a block expression to stop
      the program on detecting end-of-file ;
```

```
      while
      begin
         Put(Error, "/H&,A/", "Do next?");
         Getcard(Input, Line);
         ¬ Filemark
      end do
      begin
         Command := Scan_Input_Text(Line);
         Process_Command(Command)
      end;

      comment Release catalog file and stop;
      Release("CATALOG");
   end Of_Program.
```

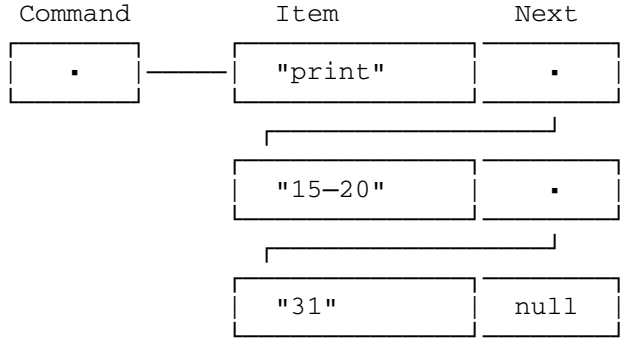
In the examples given for the predeclared procedures Xgetcard, Xputcard and Xdelete in the section "Stream Directed Input and Output", use was made of the idea of a catalog file maintained by an Algol W program. The example program above shows how a simple command interface could be built for such a program; the actual file manipulation routines are not shown.

The advantage of providing a command interface for such a program is that the system so created becomes much more friendly to the user. Rather than a data input error causing the program to stop with a fatal Algol W run error, a clear message can be printed allowing the user to retype the command for a second attempt. The additional Iocontrol options previously discussed in this section allow simple command scanners to be built with relative ease. They are, however, no substitute for the true parsing algorithms required for more sophisticated text processing programs.

If the command:

```
print 15-20 31
```

is entered the procedure Scan_Input_Text will cause the following data structure to be returned:



Getstring is used to break down the command string into its constituents. The Iocontrol keywords redefine the string recognition algorithm (to block quote and prime delimiters) and allow Getstring to resume scanning after processing each "word" in the command string.

The procedure Process_Command uses a Case statement to check the first element of the list for a possible command string. Note the use of the Translate predeclared procedure to convert the string to upper case. Translate and Uppercase are described later in this section.

If a command is identified, the relevant command procedure is called. Each such procedure is passed the command parameters as the remainder of the linked list.

This procedure demonstrates certain features of structured programming:

(1) There is a separate procedure for each logically separate algorithm.

(2) There are no Goto statements - these could easily destroy the logical flow through the program making it unreadable to others.

(3) It has been coded so that it could easily be extended.

    Extra commands could be added by adding further blocks to the Case statement.

    Command abbreviation processing could be added by replacing the individual string comparisons by calls to a logical function which checked for minimum allowed substrings.

(4) The detailed comments are not included just to inform readers of this manual. It is always good practice to include extensive comments within programs. Such documentation can be an invaluable reminder both to the author and to others who may have to modify the program later.

## OBTAINING LENGTHS FOR STRING INPUT

Occasionally it is useful to know how many characters were input into a string. Algol W provides an aid to the programmer for this purpose. When strings are read in free format, that is:

(1) using Read or Readon, or

(2) using Get, Geton or Getstring with the format parameter given as the reference constant 'null',

the value of the predeclared integer R0 is set to the number of characters moved into the string. This number does not include blanks added on the right to fill the supplied string or substring.

For example:

```
begin
   string(80) Str;
   integer N, Length;
   Read(N);
   for I := 1 until N do
   begin
      Read(Str);  Length := R0;
      Write(I_W := 2, Length, Str)
   end
end.
```

If the following data are supplied:

```
5
These
are
"some character strings"
"to   "
'be read'
```

the program will print:

```
 5  These
 3  are
22  some character strings
 5  to
 7  be read
```

Note that on the fourth line of output the explicitly specified trailing spaces have been included in the character count.

## STOPPING AN EXECUTING PROGRAM

Algol W provides the predeclared procedure Stop to immediately terminate execution. The manner of termination is exactly as if normal end of program had been passed. The form of the procedure call is:

```
Stop(<expression>)
```

where <expression> may be any simple expression. However, if the expression is a string it will be printed on the ERROR stream before the procedure stops program execution.

For example:

```
if No_More_Data then Stop(null);
```

This If statement will cause normal program termination if the value of the logical variable No_More_Data is 'true'.

## TRAPPING ATTENTION INTERRUPT CONDITIONS

Algol W provides the predeclared procedure Attntrap to allow a program to intercept attention interrupt conditions generated when an interactive user presses the break key on a conversational terminal. The form of the procedure call is:

```
Attntrap(<logical-expression>)
```

If an expression of simple type other than logical is given the procedure call will be ignored. The call

```
Attntrap(true)
```

enables trapping and sets Attnmark to 'false'. When an attention interrupt occurs the predeclared logical variable Attnmark will be set to 'true'. Execution of the program continues unchecked and the trap remains set. It is the user program's responsibility:

(1) to check the value of Attnmark at convenient points within the program logic for the occurrence of an attention interrupt and act accordingly;

(2) to reset Attnmark to 'false' when an attention has been processed so that the logic may test for subsequent attentions.

The call

```
Attntrap(false)
```

resets the trap. This returns to the default case where attentions are handled by MTS. Program execution will be halted by an attention interrupt. It may be resumed by issuing the MTS command:

```
$RESTART
```

See MTS Volume 1, The Michigan Terminal System, for details of the optional parameters available with $RESTART.

## EXTENDED STORAGE ACCESS

Warning: The facilities described under this heading are intended for experienced users needing sophisticated techniques. To be used successfully a thorough understanding of data representation on System/370 type machines is necessary. Certainly, there should be no need for undergraduate students to use these entries in the course of initial programming projects.

Three predeclared functions and five predeclared procedures are provided to allow extended access to the machine's main storage. Their purpose is to allow the use of certain machine facilities directly when otherwise assembler subroutines would have to be coded to achieve the same effect. The routines provided are:

External

> This predeclared function provides the address of an externally defined symbol. It can be used either to locate an external subroutine for use with Call, or to provide access (together with Fetch and Store) to external data areas such as FORTRAN named COMMON sections.

Halfword

> Halfword returns a halfword representation of its integer argument in an integer variable. This could be used to supply a halfword parameter to an external subroutine.

Fullword

> Fullword returns a fullword value converted from its integer argument in an integer variable, with the assumption that the leading two bytes of this integer represent a halfword. This could be used to recover an integer value from a halfword result returned by an external subroutine.

Locate

> This predeclared procedure will return the address of an Algol W variable. It provides the same facilities for Algol W as the MTS ADROF subroutine provides for FORTRAN users.

Move

> This predeclared procedure transfers bytes of data from one Algol W variable to another without any data conversion, that is it provides a straight copy operation regardless of the data type. It can provide, by a copy operation, some of the flexibility of the FORTRAN EQUIVALENCE feature.

Fetch

This predeclared procedure transfers data from a region specified by a machine address (such as that returned by External) to an Algol W variable. It is the converse of the next procedure, Store.

Store

This predeclared procedure is the converse of Fetch. Data is transferred from an Algol W variable to a region specified by a machine address.

Translate

This predeclared procedure provides the facilities of the System/370 translate instruction. Data bytes in an Algol W variable may be converted in situ by information contained in a translate table specified either as a string(256) variable or by a machine address.

When using Move, Fetch, Store and Translate, extreme care must be taken to see that only valid data lengths and addresses are given. Normal Algol W run-time error checking does not apply to these routines. As a result erroneously specified parameters may cause a program interruption. This will manifest itself as a fatal run error in the 59xx series.

## External

External is a predeclared function returning an integer result which is the machine address of the external symbol supplied as its parameter.

The general form of the function call is:

External(<esdname>)

where <esdname> is a string constant giving the external symbol name as described under Call in the section "External Linkages." A 24-bit address is returned as the integer result of the function call; the top byte of the integer is zero. This means that integer arithmetic may be done safely on the returned address in order to compute related addresses.

Example:

```
integer Addr;
 .
 .
Addr := External("EBCASC");
```

This example shows how to obtain the address of the MTS case conversion table, EBCASC. This may be used, together with the Translate predeclared procedure, to convert text held in an Algol W string from EBCDIC to ASCII representation.

## Halfword and Fullword

Occasionally, when calling external subroutines, parameters are required which must be specified as halfword (or short) integers. Algol W does not support short integers as a data type. In the past this meant that such parameters had to be dealt with in a different manner involving either 16-bit shift operations or scaling by an integer factor of 65536. To ease the call of subroutines with such parameters, two additional predeclared procedures, Halfword and Fullword, are supplied.

The general form of the function calls are:

```
Halfword(<integer-expression>)
Fullword(<integer-expression>)
```

These calls both represent integer expressions. Both of these operations are performed within the machine by 16-bit arithmetic shift operations which preserve the sign of the argument in the function result.

In the Halfword function the 16-bit shift operation is to the left. This means that the number is scaled up so that the leading two bytes of the resulting integer become a halfword representation of the fullword (or normal Algol W) integer which was supplied as the function argument. The Halfword function can therefore be used to supply a halfword parameter to an external subroutine as it is called.

Example:

```
integer Length;
string(80) Card;
 .
 .
Length := 80;
Call("SPUNCH", Card, Halfword(Length), 0);
```

In this example the MTS system subroutine SPUNCH is being called. SPUNCH requires as its second parameter a halfword integer which is the number of bytes to be written out. The function Halfword is used to supply this by converting the length contained in the integer variable Length. Note that this example is intended to demonstrate the use of Halfword; within Algol W this particular operation can be more easily achieved using the Putcard predeclared procedure.

In the case of Fullword the 16-bit shift operation is to the right. This means that the number is scaled down. If the leading two bytes of the function argument represent a halfword integer then the function result would contain a fullword (or normal Algol W) representation of the number. Thus if an external subroutine returns a halfword value then an Algol W integer variable can be supplied as the parameter to receive this. Subsequently Fullword may be called to recover an integer Algol W value corresponding to the halfword so returned.

Example:

```
integer Length, Lenparm, Lnum;
string(256) Inrec;
 .
 .
Length := 80;
Call("SCARDS", Inrec, Lenparm, 0, Lnum);
Length := Fullword(Lenparm);
```

In this example the MTS system subroutine SCARDS is being called. SCARDS requires as its second parameter a halfword integer in which it will return a value giving the number of bytes which have been read in. The Algol W integer Lenparm has been supplied to receive this. The subsequent call of the Fullword function has been used to recover the returned value so that it may be processed normally within the Algol W program. Note that this example is intended to demonstrate the use of Fullword; within Algol W this particular input operation can be more easily achieved using the Getcard predeclared procedure.

Because Halfword and Fullword use arithmetic shift operations rather then logical ones the sign of the argument is preserved in the result. However this does mean that, in the case of Halfword, a scaling up of the number could produce an integer overflow condition. This will occur if the argument is greater than 32767 or less than -32768. These values represent the limits of the range of halfword integers which can be represented on the machine.

## Locate

This predeclared procedure is used to return the address of an Algol W variable, or the address of the result of the evaluation of an Algol W expression.

The general form of the procedure call is:

```
Locate(<item>, <address>)
```

where:

<item> is the Algol W entity whose address is required; and

<address> is an integer or bits variable which will receive the address of <item>.

For this predeclared procedure to be useful, <item> will normally be specified as a simple variable identifier, a subscripted variable or a substring. If <item> is given as an expression then Algol W will evaluate this expression and return the address of a temporary region in which it is holding the result.

As with External the value returned is a 32-bit quantity of which the top byte is zero and the low three bytes give a 24-bit address.

Example:

```
long real Z;
integer array Parlist(1::8);
 .
 .
Parlist(1) :=  ......;
Locate(Z, Parlist(2));
 .
 .
Parlist(8) := ......;
Locate (Parlist(1), R1);
Rcall("QSUB");
```

This example shows how a subroutine with a nonstandard S-type calling sequence can be called from Algol W using Locate and Rcall. QSUB is assumed to be a subroutine of the kind which would normally be accessed by Call but where one or more of the parameters is not specified by address in the usual way. In such a case an integer array could be used to specify the parameter list. For each parameter of the subroutine, the corresponding element of the array Parlist must be set up. This could be done either by an ordinary assignment, if a parameter value was required within the list, or using Locate if the address of the parameter is required, as would normally be the case. These two alternatives are shown by the first and second parameters for the subroutine QSUB in the above example. When the parameter list has been completely set up, it is necessary to place the address of the first word of the parameter list in machine general register one before the subroutine is called. This is done by using Locate to store the address of Parlist(1) into the predeclared integer variable R1. Rcall loads machine general register one from R1 as it calls QSUB.

Move

This predeclared procedure is used to copy bytes of data from one Algol W variable to another.

The general form of the procedure call is either of:

```
Move(<source>, <destination>)
Move(<source>, <destination>, <length>)
```

where:

<source> specifies the Algol W variable from which data bytes are to be copied;

<destination> specifies the Algol W variable which is to receive the data; and

<length>, if supplied, is an integer expression specifying the number of bytes of data which are to be transferred by the procedure call.

<source> and <destination> represent variables of any Algol W simple type, and the transfer of bytes of data is a simple copy operation - no data conversion of any kind is done.

If <length> is omitted then the number of bytes of data moved by the procedure call is the minimum of the implied lengths of <source> and <destination>. This length will depend on the simple type of each parameter as declared, as shown in the following table:

| Simple type | Implied Length |
|---|---|
| integer | 4 |
| real | 4 |
| long real | 8 |
| complex | 8 |
| long complex | 16 |
| logical | 1 |
| string(n) | n |
| bits | 4 |
| reference | 4 |

If a subscripted variable is specified then the length is that of the simple type of the array. If a substring is given then the length of the parameter is the designated substring length.

When a length is given this may be any positive integer. Move may then be used to specify an operation which can copy beyond the bounds of either <source> or <destination> or both. For instance, several integer array elements could be copied into a string of suitable length in one operation.

Move provides an experienced Algol W user with the kind of facilities available to a FORTRAN programmer through that language's EQUIVALENCE statement. In Algol W there is no way in which two different variables may be declared to name the same region of storage so that an assignment to one also changes the other. When this facility is required, Move must be used to copy the required bytes of data from one variable to another.

Example:

```
string(80) Card;
string(4) Owner;
string(6) Volume;
string(24) Filename;
integer array Cinfo(1::16);
integer array Finfo, Sinfo(1::2);
 .
 .
 .
Readcard(Card);

Cinfo(1) := 16; Finfo(1) := Sinfo(1) := 0;
Call("GFINFO", Card, Filename, 1, Cinfo(1), Finfo(1), Sinfo(1));
if R_Code = 0 then
begin
   Move(Cinfo(3), Owner);
   Move(Cinfo(4), Volume, 6);
   Write("File: ", Card);
   Write(" ");
   Write("Owner is ", Owner);
   Write("Resides on disk ", Volume);
   Write("Has been referenced ",I_W := S_W := 1,Cinfo(6),"times");
end;
```

The above example shows a section of an Algol W program calling the MTS system subroutine, GFINFO to obtain information about a filename which is read into the string Card.

GFINFO is typical of system subroutines in operating systems like MTS which are intended to be called by such languages as Assembler and FORTRAN which do not enforce strict separation of data types.

When calling this subroutine, its specification (in MTS Volume 3, System Subroutine Descriptions) specifies the CINFO parameter as the location of a 16 element array in which catalog information about a file will be returned. Inspection of the list of data which may be returned as catalog information shows that this includes such differing types as the file owner (4 characters), the disk volume on which the file resides (6 characters) and the number of times the file has been accessed (an integer). While the final parameter can be retrieved easily from the corresponding element of the integer array on return from GFINFO, the file owner and disk volume are best dealt with in Algol W as strings. The data is already represented correctly in main storage but the data

type is inconsistent with Algol W's view of the contents. The solution shown above is to use the Move predeclared procedure to copy the contents of the relevant array elements into Algol W string variables.

## Fetch

Fetch is a predeclared procedure like Move with the important difference that data is transferred from a region specified by a machine address to an Algol W variable. It is intended to allow a transfer of information from system tables or external data areas such as a FORTRAN named COMMON section. In this context it would be used in combination with the predeclared function External, which will return the address of such an area.

The general form of the procedure call is either of:

```
Fetch(<source-address>, <destination>)
Fetch(<source-address>, <destination>, <length>)
```

where:

<source-address> is an integer or bits expression specifying the machine address from which data is to be moved;

<destination> is an Algol W variable to which bytes of data are to be transferred; and

<length> is an optional integer expression specifying the number of bytes of data to be transferred by the procedure call.

If the <length> parameter is omitted then the number of bytes moved is the implied length of the <destination> variable. A table of implied lengths for the different simple types of Algol W variable is given in the previous section on Move.

If a subscripted variable is specified, then the length is that of the simple type of the array. If a substring is given then the length of the parameter is the designated substring length.

When a length is given this may be any positive integer. A copy operation may then be specified using Fetch which can copy beyond the bounds of the <destination>. For instance, if <destination> specified the first element of an array, this could be filled with bytes from a suitable address within a FORTRAN COMMON section in one operation, provided that the <length> expression gave the number of bytes in the array.

Fetch, together with the converse predeclared procedure Store described in the next section, provides an Algol W programmer with the ability to control directly the storage management of externally held variables or arrays of variables.

Example:

```
integer Address;
long real array Data(1::100);
 .
 .
Address := External("COMBLK");
Fetch(Address + 8, Data(1), 800);
```

This partial program is representative of the manner in which bytes of data may be copied from a FORTRAN named COMMON block to Algol W variables. If the FORTRAN routine had a declaration of:

```
      INTEGER*4 NUM1, NUM2
      REAL*8 DATAVC
C
C  DECLARE NAMED COMMON BLOCK
C
      COMMON /COMBLK/ NUM1, NUM2, DATAVC(100)
```

then the Fetch statement in the Algol W program segment would copy all of the elements of the FORTRAN array DATAVC into a corresponding Algol W array Data.

## Store

Store is a predeclared procedure which is the converse of Fetch. It transfers data from an Algol W variable to a region specified by a machine address. This allows the transfer of information into external data areas such as FORTRAN named COMMON sections.

The general form of the procedure call is either of:

```
Store(<source>, <destination-address>)
Store(<source>, <destination-address>, <length>)
```

where:

<source> is an Algol W variable from which bytes of data are to be transferred;

<destination-address> is an integer or bits expression specifying the machine address to which data is to be moved; and

<length> is an optional integer expression specifying the number of bytes which are to be to be transferred by the procedure call.

If the <length> parameter is omitted then the number of bytes moved is the implied length of the <source> variable. A table of implied lengths for the different simple types of Algol W variable is given in the earlier section on Move.

If a subscripted variable is specified then the length is that of the simple type of the array. If a substring is given then the length of the parameter is the designated substring length.

When a length is given this may be any positive integer. A copy operation may then be specified using Store which can copy from beyond the bounds of <source>. For instance a complete FORTRAN array in a COMMON section could be filled in one procedure call by specifying a suitable address within a COMMON section as the <destination-address>.

Store, together with the converse predeclared procedure Fetch, described in the previous section, provides an Algol W programmer with the ability to control directly the storage management of externally held variables or arrays of variables.

For example, in the environment of the partial Algol W program used as an example in the previous section on Fetch, the statement:

```
Store(Data(1), Address + 8, 800);
```

would fill the array in the FORTRAN COMMON block with the contents of the Algol W array Data.

## Translate

This predeclared procedure provides the facilities of the System/370 translate instruction. Data bytes in an Algol W variable may be converted in situ by information contained in a translate table.

The general form of the procedure call is either of:

```
Translate(<string-variable>, <table>)
Translate(<string-variable>, <table>, <length>)
```

where:

<string-variable> specifies the Algol W string which is to be translated;

<table>, which controls the action of the procedure, may be given either as a 256-byte string expression or as an integer or bits expression specifying the address of a 256-byte translate table;

<length> is an optional parameter specifying the number of bytes which are to be translated.

If <length> is omitted the implied length of the first parameter is used, as given in the table in the section describing the Move predeclared procedure.

If a subscripted variable is specified then the length is that of the simple type of the array. If a substring is given then the length of the parameter is the designated substring length.

The action of Translate is as follows. The <string-variable> is converted in situ, that is the result of the translation replaces the original values of the characters in the string. For each character which is to be translated, its integer value is used to fetch a character from a position in the table which is offset from the start of the table by that integer value. This fetched character then replaces the original one in the string being translated. Integer values corresponding to string characters are given in Appendix B.

For a string Chars the operation is equivalent to the Algol W code:

```
integer Temp;
string(80) Chars;
string(256) Table;
 .
 .
for I := 0 until 79 do
begin
   Temp := Decode(Chars(I|1));
   Chars(I|1) := Table(Temp|1)
end;
```

The equivalent operation using Translate would be:

```
 Translate(Chars, Table);
```

As well as being more concise to code, the operation is much more efficient because it uses the computer's translation hardware directly.

Example:

```
integer Ebcdic_To_Ascii;
 .
 .
Ebcdic_To_Ascii := External("EBCASC");
Translate(Output_Text, Ebcdic_To_Ascii);
```

In this partial program the string Output_Text is being translated from the IBM EBCDIC character encoding to the ASCII variety. To do this the programmer has used the External predeclared function to locate the MTS system table provided for this task.

See MTS Volume 3, System Subroutine Descriptions, for details of the entry EBCASC and its inverse ASCEBC. See also the next section for descriptions of two translate tables provided by Algol W for case conversion.

Predeclared Translate Tables - Lowercase and Uppercase

Two predeclared bits variables are provided by Algol W to provide case conversion in either direction when used in conjunction with the Translate predeclared procedure:

bits Lowercase

This variable contains a pointer to a 256 byte translation table. When used with Translate it will convert all upper case characters to lower case leaving all other characters unchanged.

bits Uppercase

This variable contains a pointer to a 256 byte translation table. When used with Translate it will convert all lower case characters to upper case leaving all other characters unchanged.

The following program:

```
begin
   string(256) Text;
   while
   begin
      Getcard(0, Text);
      ¬ Filemark
   end do
   begin
      Translate(Text, Lowercase);
      Putcard(1, Text)
   end
end.
```

would read in the file on stream zero, translate the text to lower case and output the result on stream one. The input text:

```
@THREE @MEN IN A @BOAT,
BY @JEROME @K. @JEROME
```

would be output as:

```
@three @men in a @boat,
by @jerome @k. @jerome
```

When writing programs which take command lines as input, it is good practice to allow any mixture of upper and lower case in the command verbs. This can be achieved by using Uppercase to translate the relevant string to uppercase before attempting recognition. An example of this has already been given earlier in this section in the Section "A Simple Command Scanner."

# ALGOL W PROGRAMMER'S GUIDE

## SYSTEM DESIGN PHILOSOPHY

A single method of access is provided to the Algol W system. The behavior of the system may be modified by parameters given when the system is invoked or by control records submitted via the main source INPUT stream or by a mixture of both. However in the simple case where an Algol W source program is to be compiled and immediately executed once only, neither control records nor parameters are required.

The Algol W system is reentrantly coded. When used it is augmented by a subroutine interface which provides communication between the system proper and the operating system in use. The property of reentrancy allows all simultaneous users of any part of the Algol W system to share the object code of the system, with a consequent reduction in the memory requirements of individual user tasks.

## ALGOL W IN MTS

The object of the Algol W system is loaded into storage automatically when Algol W is run. The Algol W system is available to users via an entry point name in the MTS low core symbol table LCSYMBOL or in *LIBRARY. User programs should not depend upon this name directly as it may change without notice. This entry point is used to load routines necessary for the execution of Algol W main programs and subroutines. The loading of these routines is done automatically by MTS and Algol W.

### Input/Output Stream Names

Within Algol W there is a predefined set of 25 input/output stream names. These are the 5 named streams INPUT, PRINT, PUNCH, ERROR, and USER and the 20 numbered streams 0 to 19. The correspondence between these stream names and the MTS logical I/O units is as follows:

| Algol W Stream Name | MTS logical I/O Unit |
|---|---|
| INPUT | SCARDS |
| PRINT | SPRINT |
| PUNCH | SPUNCH |
| ERROR | SERCOM |
| USER | GUSER |

The Algol W stream numbers 0 to 19 correspond to the MTS logical I/O units 0 to 19.

## *ALGOLW

The Algol W system is accessed in MTS via the public file *ALGOLW.

```
$RUN *ALGOLW [I/O-assignments] [PAR=parameters]
```

The <I/O-unit-assignments> should specify keyword parameters of the form:

```
<unit>=<file-or-device-name>
```

where <unit> is one of the 25 valid MTS logical I/O unit names. Compiler source input including control records is read in from SCARDS. Any compiler source listing output is written to SPRINT. Any object deck will be produced on SPUNCH. Algol W system diagnostic messages appear on SERCOM.

Parameters to *ALGOLW are taken as parameters to be applied to the Algol W compiler within the system. They are fully described in the section "Compiler Parameters" later in this section.

## BASIC USE OF THE SYSTEM

The Algol W system can be invoked in several ways to compile programs:

(1) It can compile a supplied source program and immediately load and execute the object program produced.

(2) It can compile a supplied source program to an object program which has a separate existence from the Algol W system. Normally this object deck as it is called, would be stored in a file until it is required. It can be run many times by loading it from the file.

The following sections give a series of recipes for use of the Algol W system. These will satisfy the needs of most users. However many variations and extensions of these recipes are possible and a full description of system control records and parameters follows later in this section.

The term "compile, load, and go mode," sometimes abbreviated to CLG, covers all use of the system in which successfully compiled programs are immediately executed.

Compiling a source program to an object deck is sometimes referred to as "production" or "deck generation" use of a compiler. The term production mode will be used in this context in the rest of this section.

## The Compile, Load, and Go Default

If the Algol W system is invoked and no system control records (explained later) or $RUN parameters are given, then by default the system will compile a single source program and, if successful, immediately execute it. For example, if the following source deck is given to MTS:

```
$RUN *ALGOLW
begin
   for I := 1 until 10 do
   Write(I, I*I, I*I*I)
end.
$ENDFILE
```

these source records will cause the embedded program source text to be compiled. Since the program is correct the compilation will be successful, the Algol W system will immediately load and execute the resulting object program. In this case it will tabulate the integer numbers 1 to 10 with their squares and cubes.

With this simple method of processing Algol W programs, no data may be input from the INPUT stream (MTS SCARDS) since this requires use of control records. However input statements within the program may fetch data from any other input stream, either predefined or user defined. If a predefined stream is specified then the MTS unit assignment for that stream must be given with the $RUN command which invokes *ALGOLW.

Example:

Assuming that the file -DATA contains:

```
2 3 4
```

then the following source text:

```
$RUN *ALGOLW 3=-DATA
begin
   integer A, B, C;
   Get(3, null, A, B, C);
   Write("Sum = ", A+B+C);
   Write("Product = ", A*B*C)
end.
$ENDFILE
```

will compile, load, and execute the Algol W program contained within it. The Get statement will fetch three integer values from stream 3 in free format. Since stream 3 has been assigned to the scratch file -DATA, the variables A, B, and C will be assigned the values 2, 3, and 4, respectively and the program will output the sum and product of these numbers.

## Compile, Load, and Go using Control Records

The Algol W system supports several control records. These are distinguished by a slash (/) in column one of the input record followed immediately by a sequence of characters which identify the particular record. Control records are only recognized when encountered in the INPUT stream (MTS SCARDS).

In this section only two control records are discussed, /COMPILE and /EXECUTE.

/COMPILE indicates that Algol W source text follows, that is a new program is to be compiled.

/EXECUTE indicates the end of an Algol W program and completes compilation and loading of the program. It then initiates execution of the loaded program.

If a program has already been loaded and executed by the effect of a /EXECUTE record then a further one will cause the program to commence execution again. This process may be repeated as many times as desired.

Any data which the program will read from the INPUT stream should immediately follow the /EXECUTE record. If /EXECUTE or /COMPILE are encountered when input data is expected, then an end-of-file condition will be signalled to the executing program as many times as are necessary to cause it to complete execution. The control record which caused this action will then be obeyed.

If a /COMPILE record is encountered while Algol W source text is being entered then the compilation of the program will be completed. If successful, the program is then loaded and executed with the proviso

that no data may be entered from the INPUT stream. In any case, when either the compilation has failed or the program so compiled has completed execution, the /COMPILE record takes effect and causes the Algol W system to be reset so that a new program compilation is started.

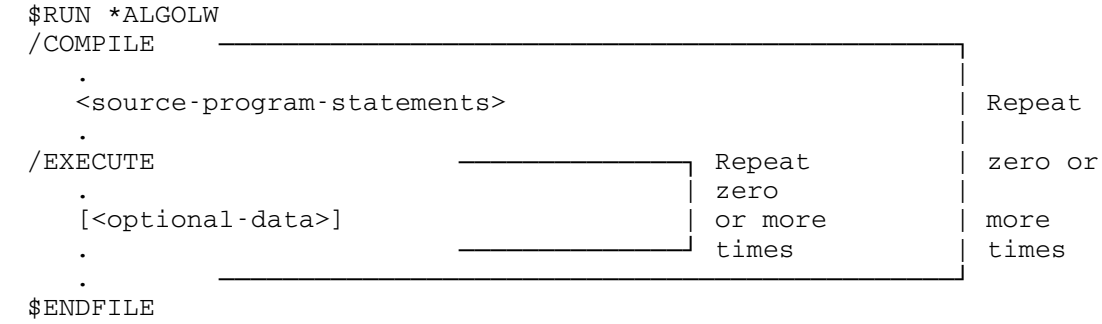
The implication of the above information is that, in one run of the Algol W system, one or many Algol W programs may be compiled and each program may be executed one or many times as desired.

Example:

```
$RUN *ALGOLW
/COMPILE
begin
   integer A, B;
   Read(A, B);
   Write("Sum = ", A+B);
end.
/EXECUTE
2 2
/EXECUTE
-45 90
/EXECUTE
10 11
$ENDFILE
```

The above sequence compiles a single Algol W program. The first /EXECUTE record completes compilation, loads the program, then starts execution. The two numbers input by the Read statement are read from the next record in the INPUT stream. Since this is all the program requires, it will print the sum via the Write statement and execution will be terminated. The program remains loaded however, so that the two subsequent /EXECUTE records and the data lines associated with them cause the program to start execution again each time and act on the two different sets of data so supplied.

So the general scheme for this simple use of control records is:




If, when the program expects to read data a /EXECUTE record is encountered instead, then end of file will be signalled to the executing program. Since the program in the example does not trap end of file

conditions, this would cause a fatal error and execution of the program would be terminated with the relevant error message being printed. The /EXECUTE then takes effect and causes execution to start afresh.

If a /COMPILE record had been encountered when the program was executing and expecting data, the same fatal termination would have occurred. However, in this case, the program would have been unloaded and the system reinitialized ready for a new program. Subsequent input records which are not control records are taken as Algol W source program text and will be treated accordingly.

## Producing an Object Deck

In production mode, an Algol W program is compiled to an object deck which has a separate existence outside the Algol W system. It would normally be kept in an MTS file. This object program may be run from the file as and when desired, provided that no change to the program logic is required. For large programs, or for programs which are to be run by many different users, it is more efficient to store the object program in a file than to compile the Algol W source text each time the program is run.

Production mode is not the default when *ALGOLW is invoked. To override a compiler default, a compiler parameter must be specified. This may be done in several ways. In this overview only two ways need concern us:

(1) they may be supplied as optional parameters on a /COMPILE record;

(2) they may be supplied in the PAR= field of the MTS $RUN command which invokes *ALGOLW.

If parameters given in the PAR= field conflict with parameters given on a /COMPILE record, that is they imply opposites, then the string in the PAR= field takes precedence. This overriding action of any parameters in the PAR= field is applied every time a /COMPILE record is encountered and processed. Control parameters on the latter normally indicate settings which are local to the next program compilation.

The important parameter for production mode is DECK.

DECK causes the Algol W system to compile any program supplied to it into an object deck. Once the compiler has started to compile a program, the mode of the Algol W system (either compile, load and go or production) is fixed for this invocation of *ALGOLW.

For programs which are successfully compiled in production mode, object cards are output. If SPUNCH is assigned, then the object modules are written to the specified file or device. If, on the other hand,

this stream is not assigned then the scratch file -AWLOAD is obtained, emptied and the object module written to it. Note that any previously existing information in this file is lost.

If production mode compilation is normally required then it is wise to store the DECK parameter on a /COMPILE record in the file containing the Algol W source program text. This both saves typing and prevents accidental loading and execution of the program.

Example:

If the MTS file PROG.S contained:

```
/COMPILE DECK
begin
   integer A;
   A := 1;
   for I := 1 until 10 do
   A := A * I;
   Write("Product = ", I_W := 1, A);
end.
```

then this program would be compiled in production mode to an object deck simply by issuing the MTS command:

```
$RUN *ALGOLW SCARDS=PROG.S
```

The object module of such a program will be written to the file -AWLOAD. It can then either be saved in a permanent file or processed further by the object file editor, *OBJUTIL (see MTS Volume 5, System Services, for a description of *OBJUTIL.

## Running Object Decks

An Algol W object program contained in an MTS file is run in exactly the same way as any other object program by using the MTS $RUN command.

Example:

```
$RUN <object-program> [<I/O-unit-assignments>] [PAR=<run-parameters>]
```

The <I/O-unit-assignments> should specify keyword parameters of the form:

```
<unit>=<file-or-device-name>
```

where <unit> is one of the 25 valid MTS logical I/O unit names. The keyword parameters set up the correspondence between input/output stream names used by the Algol W program and the file or device to which they are to be attached. The correspondence between Algol W internal stream

names and MTS logical I/O units has already been given in the Section "Input/Output Stream Names" earlier in this section. The program given in the previous section illustrated the compilation of an object deck into the scratch file -AWLOAD. This can be run as follows:

```
$RUN -AWLOAD
```

in which case the program will produce a single a line of output on the Algol W PRINT stream:

```
Product = 3628800
```

Alternatively the results of this program could be sent to a scratch file -PRINT by issuing the following MTS command:

```
$RUN -AWLOAD SPRINT=-PRINT
```

Any other program containing input/output statements specifying other streams can be run in a similar way.

Note that no special processing takes place for records input via the Algol W stream INPUT (MTS SCARDS) when the program is run from an object deck in a file; that is, no control records are recognized during this mode of execution. If any are encountered, they will be passed to the executing program as data.

If an Algol W program specifies external subroutines or precompiled Algol W procedures then the object or library file name specifying the location of those subroutines or procedures may be concatenated with the name of the Algol W object file name in the usual way.

For example, an Algol W program stored in MTS file PLOTPROG and which invokes the MTS Integrated Graphics package, can be run in the following way:

```
$RUN PLOTPROG+*IG
```

Parameters supplied in the PAR field when an Algol W object deck is run are referred to as run-time parameters. These are a different set from those expected by *ALGOLW; they are described in the Section "Run-Time Parameters" later in this section. None of them are mandatory.

## Basic Compiler Parameters

This section describes a small subset of the compiler parameters available. All parameters are described fully in the Section "Compiler Parameters" later in this section. DECK, which abbreviates to D, has already been described in the Section "Producing an Object Deck." It requests the production of an object deck.

The parameters SLIST (abbreviation S) and NOSLIST (abbreviation NOS) provide basic control over the production of a source program listing. *ALGOLW will, by default when run in MTS batch mode, write a source listing to SPRINT. This can be suppressed by using the NOSLIST parameter.

When run at a conversational terminal *ALGOLW will not produce a source program listing by default. If one is desired the parameter SLIST should be given. The compiler source listing would be written to SPRINT if that stream is explicitly assigned, that is, if SPRINT is the subject of an input unit assignment on the $RUN command which invokes *ALGOLW. If not, then the scratch file -AWLIST is obtained, emptied and used for the purpose. Any previously held information in this file will be lost.

For large programs, it is often useful to have a cross reference listing of the identifiers used in the program. With *ALGOLW this can be produced by specifying XREF (abbreviation X) as a compiler parameter. Note however that a cross reference listing will only be produced if a source listing is also produced. The two listings are written to the same file or device, one immediately after the other with the source listing first.

In numerical work, calculations on floating point quantities are often performed to the full 64-bit precision. This means declaring all floating point variables as 'long real' or 'long complex', remembering to specify the long precision versions of predeclared functions and appending a trailing 'L' to each floating point constant. This task can be somewhat laborious. The parameter LONG can achieve all of this: it causes all floating point declarations and references to be taken as referring to 64-bit precision quantities.

## Building a Precompiled Procedure Library

Frequently procedures are compiled separately from a main program. This is done either because the main program is becoming so long that it is unwieldy, because several different main programs need to share a set of procedures, or so that programs written in languages other than Algol W (e.g., FORTRAN, Assembler) may call Algol W routines. The object code for several precompiled procedures may be stored conveniently in a private library in the user's file space. They are then available on request without the overhead needed to compile them with each main program.

When *ALGOLW is run in production mode, several precompiled procedures may be compiled to an object file from one invocation of the compiler system. The Algol W source text and control records should be arranged as follows:

```
/COMPILE DECK
procedure One( ......):
begin ....
end.
/COMPILE
procedure Two( ......):
begin ....
end.
/COMPILE
procedure Three( ......):
begin ....
end.
```

Assuming this source was contained in the MTS file PROCSOURCE, all of the procedures can be compiled to the scratch file -AWLOAD by the single MTS command:

```
$RUN *ALGOLW SCARDS=PROCSOURCE
```

Note that the DECK parameter is necessary on the first /COMPILE record only. Once production mode is established by the compilation of the first procedure, the mode in which *ALGOLW runs cannot be changed and therefore the parameter need not be specified again. Procedure libraries are built or changed using the MTS object file editor program *OBJUTIL. The object modules of the procedures previously compiled into the scratch file -AWLOAD could be formatted into a library file PROCLIB using the following MTS command:

```
$RUN *OBJUTIL SCARDS=-AWLOAD 0=PROCLIB PAR=LIBRARY
```

Note that this command will set up a library in an empty file, add modules for new procedures to an existing library, or replace the object of existing precompiled procedures with new object modules for changed versions of these procedures.

Object module libraries set up using *OBJUTIL contain special directory records recognized by the MTS loader, which then loads only the object code of those procedures referenced by a main program. This provides an efficient way of storing the object code for a set of precompiled procedures in a single file without the overhead of loading all of the object code every time the library is referenced.

An Algol W main program or a program written in any language that uses the O/S Type I calling sequence (e.g., FORTRAN, Assembler) in an MTS file MAIN could reference procedures in a library PROCLIB simply by concatenating their names:

```
$RUN MAIN+PROCLIB
```

Full details of *OBJUTIL may be found in MTS Volume 5, System Services.

## CONTROL OF THE SYSTEM

This section gives full information on control of the Algol W system, that is *ALGOLW, through the use of control records. The control records available are:

| Control Record | Function |
|---|---|
| /COMPILE | enters compilation mode |
| /EXECUTE | starts program execution |
| /EOF | issues a 'soft' end of file |
| /STOP | completes compile and execute step and exits system |
| /COPY | copies source text or data from a file |
| /GLOBAL | supplies global default parameters |
| /LIST | lists subsequent source text |
| /NOLIST | does not list subsequent source text |
| /INDENT | specifies the source listing indent level |
| /NOINDENT | suppresses source listing indentation |
| /SPACE | controls white space on the source listing |
| /EJECT | skips to a new listing page retaining title |
| /TITLE | skips to new listing page and changes the title |
| /CMD | issues a system command |
| /EDIT | invokes system editor |
| /COMMENT | comments the source deck |
| /MESSAGE | prints a line of text |
| /FLUSH | completes the compile and execute step |
| /MONITOR | not used at UM |

An end-of-file indication to *ALGOLW behaves in the same way as a /STOP control record.

These control records are described in detail in the following sections, grouped according to the system context in which they apply.

The control record identifiers may be abbreviated to their first three characters including the slash (/) except for three:

| | |
|---|---|
| /COMMENT | /COMM |
| /COMPILE | /COMP |
| /NOINDENT | /NOI |

The control records /MONITOR and /FLUSH are valid during a normal run of *ALGOLW, but users would not normally be expected to use them.

## System States

The Algol W system operates in three distinct states. All control records are valid at any time when entry from the INPUT stream is possible, regardless of the current state. The three states are:

### Compilation State

Compilation state is the initial one when *ALGOLW is invoked. Once left it can only be reentered by a /COMPILE record. Any input record which is not a control record is assumed to be Algol W source text and will be sent to the compiler for processing. The following control records affect the format of any source program listing. While these are valid at all times, they are only operative in compilation state.

```
/INDENT
/LIST
/NOINDENT
/NOLIST
/EJECT
/SPACE
/TITLE
```

They are fully discussed in the Section "Compiler Source Listing Control" later in this section.

### Execution State

During execution state a previously compiled program is being executed. If a read request is pending from the INPUT stream, then this is the result of the execution of an input statement from within the loaded object program. Any record other than a control record will be taken as data to the program and sent to the run-time support routines for processing. Any control record is valid at this point.

Execution state is entered either as a result of an explicitly supplied /EXECUTE record or by one of the following control records causing the system to leave compilation state:

```
/COMPILE
/FLUSH
/STOP     (or end-of-file)
```

However data may only be read from the INPUT stream by the loaded program if execution state was entered via an explicit /EXECUTE record. The other three enter execution state only if the program is successfully compiled and then only as a step in the progression to a different state. For this reason, if programs executing as a result of their action request input, the Algol W system always returns an end of file indication to the program.

Dormant State

The Algol W system is said to be dormant if a record other than a control record read from the INPUT stream would not be sent either to the compiler or the run-time support routines. In this state the system is expecting control records which are either state independent, for example /EDIT, or a /COMPILE record which would reenter compilation state, or a /EXECUTE record which would enter execution state. /COMPILE is always valid. /EXECUTE will only be obeyed if a loaded object program exists as a result of a previous compilation. If this is not true then a /EXECUTE cannot be obeyed and the system remains in dormant state.

## Compiler Source Listing Control

Compiler listings are produced under the overall control of the compiler parameters SLIST, NOSLIST, and FULLSLIST. These are fully described in the Section "Compiler Parameters" later in this section.

If a compiler source listing is being produced then the following control records can change the format of the listing locally.

/LIST and /NOLIST enable or disable the listing of subsequent compiler source text.

/INDENT and /NOINDENT locally change the degree of automatic source text indentation for subsequent source input.

/TITLE and /EJECT both cause the listing to skip to the head of the next printer page. /EJECT will not alter the current title setting displayed at the head of the listing. /TITLE sets or resets the title string in addition to the page skip.

/SPACE leaves white space on the listing under the control of parameters supplied on the same record. As well as skipping a given number of blank lines on a listing, it may also be used to skip to the head of a new page if less than a specified number of lines remains on the current listing page.

Full desriptions of these control records are given in the Section "System Control Records" later in this section.

## Including Source or Data from Other Files

The /COPY control record takes a filename as its single parameter. When it is encountered in the INPUT stream, /COPY causes the Algol W system to suspend reading from that stream and fetch subsequent input

from the file name specified. When end-of-file is detected, the file is released and subsequent source input is fetched from the INPUT stream again. Since /COPY is valid at any time, this record may be used to include either program source text or data records requested from the INPUT stream.

Currently /COPY requests may not be nested. Also, for reasons of clarity the main control records /COMPILE, /EXECUTE, /STOP, /EOF and /FLUSH are not valid in text supplied from a file name invoked by /COPY. When these control records are encountered in this way, a warning message is printed and they are ignored.

/COPY may be indirectly specified in the form of a compiler parameter FILE, in which case the Algol W system is told to fetch a complete source program from the specified file. When end of file is detected for the included file, the system will attempt to compile and load the program. It will not initiate execution, rather it leaves the system in dormant state so that a subsequent /EXECUTE may start execution as and when desired.

For example, the following /COMPILE record:

```
/COMPILE FILE=PROG
```

is exactly equivalent to the set of three records:

```
/COMPILE
/COPY PROG
/EOF
```

## SYSTEM CONTROL RECORDS

This section gives a complete alphabetical list and description of the 19 system control records recognized by the Algol W system. All control records start with a slash (/) character in column one, which is immediately followed by the record name. If the record has parameters these are always optional with the exception of /COPY. Parameters on control records are separated from the control record name and any other parameters by one or more spaces or by a comma.

With the exception of /COMPILE, all parameters to a particular record must be included in the same input record as the name. /COMPILE may be continued across subsequent input records, each of which must start with slash-plus (/+) as the first two characters.

/CMD [<system-command>]

/CMD is used to pass system commands to the operating system. Any sequence of characters given as a parameter on a /CMD record is passed to the operating system for processing. For example in MTS:

```
/CMD CREATE DATA001 SIZE=5P
```

would cause the command to be passed to MTS to create the file DATA001.

Note that any MTS command may be given on a /CMD record but that the following five:

```
$RUN        $RERUN
$DEBUG      $UNLOAD
$SIGNOFF
```

since these will cause the Algol W system to be unloaded.

/COMMENT [<any-text>]

/COMMENT is provided to allow the documentation of card decks. This control record is otherwise ignored.

Example:

```
/COMMENT TEXT OF THE FIRST PRECOMPILED PROCEDURE
```

If this record is encountered in the source INPUT stream then it is ignored: the system will prompt immediately for another record.

/COMPILE [<compiler-parameters>]

This control record causes the system to enter compilation state where any subsequent records which are not control records will be assumed to be Algol W source text and passed to the compiler accordingly. Its action proceeds in the following order.

(1) Any Algol W program which is currently being entered is ended and the compilation phase completed. If the compilation is successful and the system is in compile, load and go mode, then:

    (a) The program is loaded.
    (b) If successfully loaded, execution state is entered with the proviso that any request for data from the INPUT stream will cause an end of file to be returned.

(2) The compiler system is reset to the default parameter settings extant on initial entry to the system.

(3) Any compiler parameters which have been stored as a result of a previously encountered /GLOBAL record are processed. This may have the effect of changing the defaults.

(4) Any parameters on the /COMPILE control record are processed. This will change the local default settings.

(5) Providing that no FILE= parameter is present, the system will prompt for more input from the INPUT stream. Continuation records for the /COMPILE are read while the first two characters of subsequent records are /+. Parameters present on these continuation records are processed as the records are encountered.

(6) When a record has been encountered which is neither a continuation record nor one of the control records /CMD, /COMMENT, /EDIT or /MESSAGE, any parameters which have been given in the PAR= field of the MTS $RUN command are then processed. These parameters are processed every time a /COMPILE record is encountered so that they override the defaults and local settings for each compilation during the current invocation of the Algol W system.

(7) Compilation state is now entered. Subsequent records which are not control records are passed to the compiler as Algol W source text. Subsequent control records are obeyed as necessary.

The control records /COMPILE, /EOF, /EXECUTE, /FLUSH, and /STOP or a physical end-of-file will cause an exit from compilation state.

Note that /COMPILE is not required as an initial record unless parameters are needed. The Algol W system is placed in compilation state when it is invoked.

If there are more compiler parameters than will conveniently fit on a single /COMPILE record, the additional parameters may be given on continuation records with a slash (/) in column one and a plus (+) in column two. For example:

```
/COMPILE SIZE=120K, SLIST=WHIZBANG.L, XREF
/+ DECK=WHIZBANG.O
/+ RUNPARM=(SIZE=96K, DATAPARM, NOCC)
```

The compiler and run-time parameters used in the above example are documented later in this section.

/COPY <file-or-device-name>

/COPY causes the Algol W system to switch its main input stream to the specified file or device. Subsequent input records are taken from that file or device until end-of-file is detected, when it is released. Subsequent input comes from the main INPUT stream again.

Example:

```
/COMPILE
/COPY W701:PROG
/EXECUTE
/COPY W701:DATA
/STOP
```

These control records compile the program whose source text is held in the file W701:PROG and then execute it supplying data from the file W701:DATA.

Note however that program text could also be supplied using the FILE= compiler parameter described later.

/COPY is intended to provide program source text or data to the Algol W system. It is not intended to supply an alternative main INPUT stream to control the system. For this reason certain control records are ignored, although a warning message is printed, when they are encountered in a file named on a previous /COPY control record. When a nested /COPY record is encountered a warning message is printed and control returns to the main INPUT stream after releasing the file or device.

For a discussion of /COPY operation see the previous Section "Including Source or Data from Other Files."

/EDIT [<file-name>] [:<edit-command>]

/EDIT invokes the system file editor. If no parameters are given editor command mode is entered exactly as if $EDIT had been issued from MTS command mode. /EDIT may also take a filename and a single edit command as its parameters in the same way as the $EDIT command. Note that the colon (:) specifying the start of an editor command is not necessary if an edit filename has been given on the same record.

Example:

```
/EDIT PROG.S
```

This control record will cause edit mode to be entered with the file PROG.S as the editor file. Subsequent input will expect edit commands until the editor terminates normally via an MTS, RETURN or STOP editor command or an end-of-file indication. When such editor

termination occurs, the Algol W system will be in exactly the same state as it was in when the /EDIT was encountered.

Example:

```
/EDIT PROG.S PRINT 352 C=8
```

This is an example of a one shot editor command. In this case it would print 8 lines of the file PROG.S starting with line 352. When this editor command has been executed, the editor returns control immediately to the Algol W system. Editor command mode is not entered.

For a full description of the MTS $EDIT subsystem consult MTS Volume 1, The Michigan Terminal System.

/EJECT

/EJECT causes subsequent source text to be listed on a new printed page within the source listing. The title string which appears at the head of each page remains unaltered.

It may be preferable to use the conditional page skip mechanism described later under /SPACE as this will generally waste less paper while still retaining tidy output.

/EOF

/EOF supplies an end-of-file indication to the Algol W system, that is it signals end-of-file to the state currently in control but does not cause the Algol W system to terminate. If the system is dormant, end-of-file has no effect.

If the system is in compilation state, /EOF sends an end-of-file signal to the compiler indicating that no more program source text follows and that compilation of the program previously entered should be completed. If this compilation is successful, the object program will be loaded. The system is then put into dormant state. A subsequent /EXECUTE control record could then initiate execution of the loaded program.

If the system is in execution state, an end-of-file indication is sent to the run-time support routines. How this is dealt with depends on the coding of the Algol W program being run. For instance, if the input request was made from a Read call and no assignment had been made to the predeclared Endfile reference, then execution of a loaded program would terminate with a fatal run error message. On the other hand, if the program trapped end-of-file conditions, then the effect would be under the control of the relevant program logic.

Note that /EOF is the only way to signal end-of-file to an executing program in compile, load and go mode which will leave the

system in dormant state with the program still loaded so that a subsequent /EXECUTE may restart execution.

/EXECUTE [<run-time-parameters>]

This control record causes the system to enter execution state where subsequent records which are not control records will be assumed to be data read by the executing program from the INPUT stream. Its action proceeds in the following order:

(1) Any Algol W source program which is currently being entered is ended and the compilation phase completed. If compilation is successful the object program is loaded in compile, load and go mode.

(2) If a loaded object program exists, execution is started. The system will now be in execution state.

(3) If no loaded object program exists then an error message is printed and the system is put into dormant state.

If a program is to be executed in compile, load and go mode and that program is to read data from the main INPUT stream, execution must be started by a /EXECUTE record because this is the only way to enter execution state in such a way that input requests prompt for data rather than returning end of file.

Any characters appearing after /EXECUTE on the record are taken to be run-time parameters and are sent to the run-time support routines for processing when the program is started. These parameters are equivalent to those supplied via the PAR= field when the Algol W object program is run from a file. They are described in the Section "Run-Time Parameters" later in this section.

/FLUSH

/FLUSH is not used at UM but is documented here for completeness. Its purpose is to put the system into dormant state with no loaded program available but, in route, it will complete all possible stages of compilation, loading and execution. It will:

(1) complete compilation of any partially entered program;

(2) load the program if compilation was successful;

(3) if a loaded program exists, which has not yet been executed, and the system is not in dormant state, it will start execution;

(4) if the program prompts for data, it will supply an end of file indication until execution terminates for this or any other reason, such as the time estimate exceeded;

(5) it will then unload the program.

After a /FLUSH record has been processed the system is effectively brought up to date. All output for the previous program has appeared and no further Algol W processing can take place until a subsequent /COMPILE record reenters compilation state. /FLUSH is therefore provided to allow resource scheduling systems to separate the output from a stream of individual jobs supplied to the system.

/GLOBAL [<compiler-parameters>]

This control record allows a user to change the default settings of compiler parameters in such a way that they may be locally overridden by parameters from a /COMPILE record. Compilation parameters are processed in the following order:

(1) parameters from the /GLOBAL record;

(2) parameters from a /COMPILE record and any of its continuations;

(3) parameters from the PAR= field of the $RUN command which invoked the system.

Example:

```
/GLOBAL SLIST,XREF
```

This record indicates that any subsequent compilations should produce a compiler source listing and identifier cross reference table unless otherwise overridden. The record:

```
/COMPILE NOXREF
```

would locally suppress the identifier cross reference for this particular compilation. SLIST would still cause a compiler source listing to be produced unless otherwise overridden by any PAR= field parameters.

If a /GLOBAL record has no parameters, any previously held global parameters are deleted. The /GLOBAL parameters which are in effect are only those from the last /GLOBAL record: the effect is not additive. If no /GLOBAL record has been encountered then no global parameters exist.

/INDENT [<indent-step>]

/INDENT sets the automatic indentation level on the compiler source listing for subsequently encountered source text. If <indent-step> is not specified it defaults to 3.

Only unsigned integers are valid. Any value greater than 5 is treated as though it was 5. When source text is indented on the

listing, the number of spaces by which it is indented is determined by multiplying the current block level, that is the 'begin'-'end' depth, by the current <indent-step>. If the value so determined exceeds 32, it is taken as 32.

Example:

```
/INDENT,5
```

This sets the <indent-step> to be 5 spaces per block level in subsequent source text.

/LIST

If the compiler parameter NOSLIST is in effect then /LIST is ignored. NOSLIST is the default at a terminal. If a compiler source listing is being produced but the listing of source text has been temporarily suppressed by a /NOLIST record (see later), then the /LIST causes subsequent source text to be listed again.

/LIST and /NOLIST can be used together to suppress those parts of a source program listing in which the user is not currently interested, in order to save both time and paper.

/MESSAGE [<any-text>]

This record is a printing version of /COMMENT. If any text appears on the record, it is printed on SERCOM; otherwise a blank line is printed.

Example:

```
/COMPILE
/MESSAGE   Gaussian Elimination Program
begin
  .
  .
  .
end.
/STOP
```

In this example the /MESSAGE record causes the character string "Gaussian Elimination Program" to be printed on SERCOM. Such records provide a means of verifying the program being compiled or of following the progress of a long compilation where several procedures are being compiled in one run.

/MONITOR

This record is provided to allow system programmers to implement student batch monitor systems. It is documented here for completeness.

/NOINDENT

This record is equivalent in action to:

/INDENT,0

It suppresses automatic indentation of subsequent program text on the compiler source listing.

/NOLIST

If the compiler parameter FULLSLIST is in effect then this record is ignored. If a compiler source listing is being produced then a /NOLIST record will disable subsequent listing of source program text. This may be left disabled until the end of the program or it may be reenabled by a subsequent /LIST record.

/SPACE [<lines> [<remainder>]]

/SPACE produces white space on the compiler source listing. It can be used to separate procedures or other groups of source program text in order to improve the readability of the source program listing.

If /SPACE is given with no parameters then one blank line is left on the source listing.

If /SPACE is given with a single parameter, this should be an unsigned integer between 1 and 20. It specifies the number of lines of white space to leave on the listing. If less than this number of lines remain on the page, a new page is forced exactly as if a /EJECT record had been encountered. If more than 20 lines is specified then the number is taken as 20.

If two parameters are given on a /SPACE record, the first is the number of lines as just described. The second is an unsigned integer between 1 and 20 which specifies the number of lines which should remain on the page after the /SPACE record has been obeyed. Otherwise a new page is forced, again exactly as if /EJECT had been encountered. This could be used to keep a number of lines of text together on the same listing page.

Examples:

/SPACE 3

This causes a skip of three lines on the listing.

/SPACE 4,10

This will skip four lines and if less than 10 lines then remain on the current listing page it will start a new one. If a /SPACE command of this form was inserted before every procedure declara-

tion then this would ensure that every procedure was separated by 4 lines from the previous program text and also that the procedure heading and body were not separated by a compiler listing page boundary, thereby improving readability.

/STOP

The purpose of /STOP is to return control to the system which invoked Algol W - usually MTS. In route it will complete all possible stages of compilation, loading and execution. It will:

(1) complete compilation of any partially entered program.

(2) Load the program if compilation was successful in compile, load and go mode.

(3) If a loaded program exists, which has not yet been executed, and the system is not in dormant state, it will start execution.

(4) If the program prompts for data, it will supply an end-of-file indication until execution terminates for this or any other reason such as the time estimate exceeded.

(5) It will then unload the program.

(6) It releases all storage and files acquired by the Algol W system and returns to the caller, normally MTS.

/TITLE [<title-string>]

If a source program listing is being produced, /TITLE causes a new page to be forced as would /EJECT. It will also set or reset the title string which appears in the center of the first printed line on each listing page.

If no parameter is given with /TITLE, the title string is reset to blank. If a parameter is given, it should be a string entered in the same format as that required for free format string input to an executing program, that is a group of characters delimited either by quotes (") or primes (') or a single group of characters containing no primes, quotes, spaces or commas. This string becomes the new title string and appears on the first line of each listing page until such time as any subsequent /TITLE resets it.

An end-of-file indication to the system at any time when a control record is expected is processed exactly as if /STOP had been entered and will therefore cause system termination.

## COMPILER PARAMETERS

Compiler parameters may be specified in three ways:

(1) on a /GLOBAL record,

(2) on a /COMPILE record, and

(3) as the PAR= field of the $RUN command which invokes *ALGOLW.

When a particular control record is encountered, parameters are scanned in the above order with evaluation taking place from left to right if more than one parameter is given in any particular location. Compiler parameters take either of the following forms:

```
<keyword>
<keyword>=<expression>
```

Keywords may normally be abbreviated down to a minimum of three characters if there is more than this number of characters in the word. In certain cases shorter abbreviations are allowed. For each keyword described, the minimum abbreviation is shown by underlining a part of the keyword. Any abbreviation from this limit up to the full keyword is acceptable.

A full description of each of the available compiler parameters is given in the following sections. The parameters are grouped into logically related sets.

### Selecting Object Deck or Compile, Load, and Go Mode

By default the Algol W system will compile and execute any source program which is supplied to it. This state is called compile, load and go mode. If an object deck is required, this must be specified by parameters which are as follows:

DECK | DECK=<filename> | EXECUTE  Default: EXECUTE

DECK specifies that an object deck is to be generated. If SPUNCH is explicitly assigned then the object module will be written to the attached file or device. Otherwise the scratch file -AWLOAD is obtained, emptied and used for this purpose.

If the second form is given, then <filename> is emptied and the object modules are written to it.

EXECUTE specifies the default case where object modules are not written out but are instead directly loaded in compile, load and go mode. This parameter can be used from the PAR= field to override a DECK parameter given on a /COMPILE or a /GLOBAL record.

Note that once a decision has been made to either generate an object deck or execute the first program supplied to *ALGOLW, this mode stays in effect for future compilations during this run. DECK or EXECUTE parameters for the second and subsequent programs are therefore ignored.

Examples:

If an object deck is normally desired for a particular Algol W source program then the following record should be the first in the source file:

```
/COMPILE DECK
```

If the object file for a particular program, which is stored in the file PGM.S, is always required in a file PGM.O then the first record of file PGM.S should be:

```
/COMPILE DECK=PGM.O
```

This ensures that file PGM.O is emptied before each recompilation writes a new object module to the file and this avoids problems which can occur if a subsequent version of a program produces a slightly smaller object deck than the previous compilation.

If a compilation fails in pass one or two, so that no object cards are produced at all, then an object file, whether implied by default or explicitly specified by a DECK=<filename> parameter, will be left unchanged. The file is not emptied until the first object record is ready to be written to it.

## Source Listing Control

By default Algol W will produce a compiler source listing in batch mode but not if run at a terminal. This default can be overridden or the form of the listing altered by the parameters given below:

SLIST | SLIST=<filename> | SLIST=* |
FULLSLIST | FULLSLIST=<filename> | FULLSLIST=* |
NOSLIST  Default: see text

SLIST specifies that a source listing is to be produced. This is the default in batch. At a terminal, NOSLIST is the default so in order to produce a source listing SLIST must be specified. If SPRINT is explicitly assigned then the source listing is written to the attached file or device. Otherwise the scratch file -AWLIST is obtained, emptied and used for this purpose.

SLIST=<filename> specifies that a source listing is to be produced and written to <filename> which will be emptied before use. An explicit assignment to SPRINT will, however, cause the listing to be written to that stream rather than the designated file.

SLIST=* specifies that the source listing is to be written to SPRINT regardless of whether the stream has been explicitly assigned or left to default by the system. This option can be used to force a source listing out onto the terminal.

The three options of FULLSLIST work in the same way as those for SLIST but additionally /NOLIST records encountered in the source will be ignored. This keyword can be used to ensure that the listing produced is complete.

NOSLIST is the default at a terminal and suppresses the compiler source listing completely. Any /LIST control records encountered in the source program are ignored.

XREF | NOXREF Default: NOXREF

XREF specifies that a cross reference listing of the identifiers used in the Algol W source program is to be appended to the source program listing. A cross reference listing will only be produced if a source listing has already been printed. If this is not true the XREF parameter will be ignored.

NOXREF suppresses the production of an identifier cross reference listing.

LINECNT=<integer> Default: LINECNT=60

This parameter specifies the number of lines per page to be assumed by the Algol W system when a compiler source listing is produced. The default of 60 defines a normal printer page. A minimum of 25 and a maximum of 100 lines is enforced.

INDENT | INDENT=<indent-step> | NOINDENT Default: NOINDENT

INDENT specifies the level by which source text will be indented to the right on a compiler source listing. <indent-step> may be an unsigned integer between 0 and 5: if it is not specified it defaults to 3.

Source text is indented on the listing by a number of spaces calculated by multiplying the current indent step by the current level of 'begin'-'end' nesting. For instance source text at the fourth block level with an indent step of 5 spaces per block level will be indented 20 spaces to the right. If the indent so calculated is greater then 32 spaces, 32 spaces are used. Any manual indentation will be preserved in addition to the auto-indentation.

NOINDENT suppresses automatic text indentation. It is equivalent to a specification of INDENT=0.

NUMBER | NONUMBER Default: NUMBER

NUMBER, which is the default, specifies that source file line numbers are to appear on the compiler source listing. They are printed to the left of the source program co-ordinates.

NONUMBER suppresses printing of these line numbers.

PSKIP | NOPSKIP Default: PSKIP

PSKIP causes a blank line to be produced on the compiler source listing before each procedure heading. It therefore separates a group of procedures on the listing.

NOPSKIP suppresses automatic procedure separation on the source listing.

The remaining parameters in this section control automatic translation of source text as it is copied to the source listing.

TRIDENTIFIER=<keyword> Default: TRIDENTIFIER=OFF

This parameter controls translation of the identifier names in the source program for display in the source program listing. <keyword> is one of the following:

(1) OFF

This is the default action.

No translation is done. Identifiers appear on the listing in the form in which they exist in the source program text.

(2) LOWERCASE or LC

Identifiers are translated to lower case for display in the the source listing.

(3) UPPERCASE or UC

Identifiers are translated to upper case for display in the source listing.

(4) MIXEDCASE or MC

Identifiers are translated to lower case, with the exception that the first character and any character immediately following an underscore character (_) are translated to upper case. For example, SUM_OF_SQUARES appears as Sum_Of_Squares.

TRRESERVED=<keyword> Default: TRRESERVED=OFF

This parameter controls translation of the reserved words in the source program for display in the source program listing. <keyword> is one of the following:

OFF | LOWERCASE | LC | UPPERCASE | LC | MIXEDCASE | MC

These keywords control the translation of reserved words in the same way as the keywords of parameter TRIDENTIFIER control that of identifier names. The MIXEDCASE option capitalizes only the first character as no reserved words contain the underscore character (_). The default is to do no translation.

ULRESERVED={ON|OFF} Default: ULRESERVED=OFF

This parameter controls underlining of reserved words when displayed in the source program listing. Use of this option will cause records with the ANSI overstrike carriage control character (+) to be included in the source listing file. To be effective the device selected to print the source listing must be capable of overstriking such lines. All IBM 1403 compatible line printers satisfy this requirement.

TRLITERAL={OFF|UPPERCASE|UC} Default: TRLITERAL=UC

This keyword controls translation of certain characters in constants when displayed on the source listing. Note: string constants are never translated. If the UPPERCASE option is in effect:

(1) In long precision floating point constants, the trailing "L" is capitalized.

(2) In imaginary constants the trailing "I" is capitalized.

(3) In bits constants the hexadecimal digits "A" - "F" are capitalized.

UC is an alternative to UPPERCASE.

TRCOMMENT={OFF|LOWERCASE|LC} Default: TRCOMMENT=OFF

This keyword controls translation of text within comments. If LOWERCASE or LC is specified, all comment text is displayed in lower case on the listing. While easy to specify, this is a rather lazy way to format comments. Better readability will be obtained if the comments are typed in mixed case, leaving TRCOMMENT to default OFF.

It should be noted that several system control records provide certain of these functions on a local basis within program source text. These are discussed in the previous Section "Compiler Source Listing Control."

Compiler Control

These parameters control the basic action of the compiler during compilation but exclude those concerned with source listing control or object deck generation.

SIZE=<integer>[{K|P}] Default: SIZE=16P

This parameter specifies the amount of main storage which will be available to the compiler during any particular compilation step. SIZE is specified as a number of bytes of storage. Optional scale factors of K (kilobytes, value 1024) or P (pages, value 4096) may be applied to the integer. A minimum of 6P and a maximum of 256P (one segment) are currently enforced. The default value of 16P (65536 bytes) is sufficient for moderately large source programs up to about 1000 statements.

Example:

```
/COMPILE SIZE=60P
```

If a program requires more than the default storage for its compilation, it is wise to include the size on a /COMPILE record with the source text.

In compile, load and go mode, storage used during compilation is used again by the loaded program. The maximum amount of storage available for data and record storage at one time is determined by the size of the compiler working storage less the size of the compiled program.

FILE=<filename>

FILE can be used on a /COMPILE record to specify a filename containing source program text to be compiled. The file specified should contain the entire program. The FILE parameter switches the compiler INPUT stream to read from this file. Subsequent source records are fetched by the Algol W system until end of file is detected. Then an end of file indication is sent to the compiler itself which causes it to complete the compilation of the program, leaving the system in dormant state.

Example:

Consider a source program stored in the file MYPGM.S.

```
/COMPILE FILE=MYPGM.S
```

The /COMPILE record is all that is required to compile this program, although, of course, other parameters may be specified as desired. The above /COMPILE record is equivalent to:

```
/COMPILE
/COPY MYPGM.S
/EOF
```

It is in fact implemented in this way, so that the restrictions on control records which apply to a file included by a /COPY control record also apply to one included via a FILE parameter.

TERSE | VERBOSE  Default: VERBOSE

By default, VERBOSE operation will print all error messages and warnings produced by the compiler. Also, pass one errors do not suppress further checking by pass two of the compiler. This default action can sometimes produce an exceptional number of error messages but the additional information may well be of use to an inexperienced Algol W programmer.

The TERSE parameter is provided for expert users who require only to be led to the site of an error rather than being given an extensive diagnostic appraisal of it. When TERSE is specified the following changes to the compiler behavior take place:

(1) Fatal, that is nonwarning, error messages in pass one terminate compilation at the end of that pass. (VERBOSE operation would cease at the end of pass two.)

(2) While all warning messages are printed, (subject to (3) and (4)), only one actual error message is printed for each source program co-ordinate at which an error is detected.

(3) A warning message is printed for only the first name parameter encountered rather than for each one in the program.

(4) A warning message is printed for only the first goto statement encountered rather than for each one found.

ECHO | NOECHO  Default: see text

ECHO is the default when running at a terminal. If ECHO is in effect, Algol W keeps a compressed form of source listing in virtual memory during a compilation. If a compiler error is detected in either pass one, two or three, this listing is interrogated to provide the lines of source listing nearest to the error.

The echoed source listing text is printed immediately before the error message. If more than one error message applies to a particular source program co-ordinate, the text will only be echoed once, before the first error message for that co-ordinate. A maximum of eight lines of source text will be echoed for a particular source co-ordinate. If a co-ordinate extends over more than eight lines, a warning message is printed after the eighth and the remainder of the source is suppressed.

NOECHO is the default in batch. When it is in effect no compressed listing is retained and therefore no source text is printed with the error messages.

## Control of Program Loading

This section describes compiler parameters which exercise control over the loading of object programs. These are mainly concerned with compile, load and go mode but the LIBSEARCH parameter applies to production mode as well.

### LIBSEARCH=<filename>

Algol W source programs may specify externally defined precompiled procedures or subroutines. The LIBSEARCH parameter allows the specification of an object file containing the object modules for these routines.

In compile, load and go mode the filenames specified are searched by the system loader to resolve undefined symbols. When an object deck is being generated, a record containing:

```
$CONTINUE WITH <filename> RETURN
```

is written as the last record of the object file. This allows the MTS loader to find the object for the external routines when the program is subsequently run.

Example:

```
/COMPILE DECK, LIBSEARCH=MYLIB+*IG
```

This /COMPILE record causes an object deck to be generated with a reference to the two library files MYLIB and *IG in the last record. Any reference to subroutines in these libraries will then be resolved by the MTS loader. This means that the program can be invoked merely by typing:

```
$RUN <object>
```

instead of:

```
$RUN <object>+MYLIB+*IG
```

Storing library filenames with the program source text is good practice because it saves typing and eliminates the risk of forgetting to specify library filenames, which would be an irrecoverable error in a batch job.

MAP | NOMAP Default: NOMAP

This parameter only applies in compile, load and go mode. If MAP is specified, a loader map of the object program is produced by the Algol W loader. If LIBSEARCH has also been specified, an MTS map will precede the Algol W map.

NOMAP suppresses printing of the loader map.

PROMPT | NOPROMPT Default: PROMPT

This parameter only applies in compile, load and go mode when running at a terminal. PROMPT specifies that, if external references are not locatable in any supplied library files (specifed by LIBSEARCH), the MTS loader should prompt the user for filenames from which the missing references may be resolved.

NOPROMPT suppresses this action: program execution will terminate with an Algol W loader error message.

If the word CANCEL is given as a reply to an MTS loader prompt, the Algol W system will continue as if NOPROMPT had been specified.

When a program is being run conversationally from stored object in a file, MTS will always prompt for the location of missing references.

## Object Program Attributes

The following compiler parameters specify attributes of the compiled object program.

LONG | SHORT Default: SHORT

In many numerical applications all floating point calculations are performed to 64-bit precision: for example, in a program which calls routines in the NAAS library, since all NAAS subroutines work to 64 bit precision. In such a program a user would have to be extremely careful to ensure all declarations, constants and functions specified the relevant long real and long complex forms.

The LONG compiler parameter makes this operation simple and safe. If it is in effect the following actions are taken:

(1) All declarations of type real are converted to type long real.

(2) All declarations of type complex are converted to type long complex.

(3) All real constants are converted to the relevant long real constants. For instance 1.234 becomes 1.234L.

(4) All imaginary constants are converted to the relevant long imaginary form. For instance 5I becomes 5IL.

(5) All references to floating point predeclared functions which specify the short form, generate references to the long form, for instance Sin refers to Longsin.

(6) Any references to the predeclared real variable Epsilon generate references to the alternative long real predeclared variable Longepsilon.

(7) All variables, constants and functions in the program which are already long real or long complex remain unchanged.

Once again, if it is essential for the correct operation of the program that the LONG parameter be specified, it should be stored with the program source text on a /COMPILE record.

SHORT, which is the default, will compile the program as it is entered. Only those variables, constants and functions which explicitly specify the long real or long complex forms will be compiled to use 64-bit precision.

RUNPARM= | RUNPARM=<run-parameter> |
RUNPARM=(<run-parameter-list>) Default: RUNPARM=

When an object program starts to execute, certain attributes of the running program may be set via run-time parameters. These are fully described in the Section "Run-Time Parameters" later in this section. Run-time parameters may be specified in two ways: either by an explicitly specified group of parameters given with the instruction which runs the program (either $RUN or /EXECUTE) or by using the RUNPARM compiler parameter. When such parameters are specified by a RUNPARM compiler parameter they are processed before those from a $RUN or a /EXECUTE record. They therefore provide a method of changing the default settings of these run-time parameters.

RUNPARM may be specified in three ways. If a single parameter is being provided then it is given as the right hand side of the RUNPARM specification.

```
RUNPARM=SIZE=80K
```

If several parameters are to be supplied, they should all be given in a single RUNPARM specification enclosed in parentheses.

```
RUN=(SIZE=160K, DATAPARM)
```

Note that the characters within the parentheses take the same form as an explicitly supplied $RUN parameter string.

To suppress completely a previously supplied RUNPARM string a null RUNPARM parameter should be given.

```
RUNPARM=
```

This is the default.

Note that the RUNPARM string so supplied will be stored in the object deck of the compiled program. The stored string is that which was given in the last RUNPARM directive encountered: the effect is not additive.

## Execution Resource Control

The following compiler parameters apply only in compile, load and go mode. They are equivalent to run-time parameters which perform the same tasks.

ETIME=<number>[{S|M}] | TIME=<integer>[{S|M}]
Default: see text

When a program executes in compile, load, and go mode, a limit may be set on the total cpu time which may be used by the program. The time limit is determined as follows:

(1) If a limit is set by the run-time ETIME parameter then this is used.

(2) If no run-time limit is subsequently set, but a compiler ETIME parameter specifies one, then this time is used. The run-time parameter overrides the compiler parameter.

(3) If no ETIME parameter of any kind is given, Algol W tries to find out if a system time limit exists, and, if so, when the program starts to execute the most immediate system limit is determined. This may be either a local time limit on the MTS $RUN command, or a global time limit on an MTS batch job. If none exists, no time limit is set.

(4) If such a limit does exist, an Algol W time limit is set which is slightly ahead of the system limit. This allows Algol W to take control before the system does so.

Using the ETIME parameter, alternative values between 0.2 seconds and 60 minutes may be specified. They are given as an unsigned number followed by an optional scale factor. If no scale factor is given, or it is S, then the quantity is taken to be specified in

seconds. If the scale factor is M, it is taken to be in minutes. The number given may be specified with up to three places of decimals.

ETIME and TIME are synonyms. They specify the execution time of a program. If this time estimate expires before the program completes execution, then execution is forcibly terminated and a fatal run error message printed. For example:

```
/COMPILE ET=1.6S
```

This record sets the time estimate for the program to 1.6 seconds. Note that this estimate could be overridden in turn by a further run-time parameter specified on a /EXECUTE record.

Time limits given as compiler parameters have no effect on the execution of a program run from an object deck.

EPAGES=<integer> | PAGES=<integer> Default: see text

When a program executes in compile, load and go mode, a limit may be set for the maximum number of pages of printed output which it may produce. The page limit is determined as follows:

(1) If a limit is set by the run-time EPAGES parameter then this is used.

(2) If no run-time limit is subsequently set, but the compiler EPAGES parameter specifies one, then this number is used. The run-time parameter overrides the compiler parameter.

(3) If no EPAGES parameter of any kind is given, Algol W tries to find out if a system page limit exists, and, if so, when the program starts to execute the most immediate system limit is determined. This may be either a local page limit on the MTS $RUN command, or a global page limit on an MTS batch job. If none exists, no page limit is set.

(4) If such a limit does exist, an Algol W page limit is set which is 1 page less then the system one. This allows Algol W to take control before the system does so.

Using the EPAGES parameter, an alternative value for the limit is specified as an unsigned integer between 1 and 200.

The page count is maintained with the following assumptions:

(1) Before any records are output, the printer is positioned on the perforations between two pages.

(2) Carriage control is in effect. All of the MTS logical carriage control characters are accepted.

(3) Printer pages are assumed to have a physical size of 66 lines, containing a 3 line header, a 60 line logical page and a 3 line trailer. Use of the ANSI standard logical carriage control characters (space, "0", "-", "1") will write into the logical page only, skipping the 6 line trailer and header from the bottom of one logical page to the top of the next.

(4) Only output to the PRINT stream (MTS SPRINT) is counted, unless the executing program specifies otherwise using a Qualify predeclared procedure call with the PAGELIMIT parameter.

EPAGES and PAGES are synonyms. If an executing program attempts to print more pages than the estimate on the streams for which pagecount checking is enabled, program execution is forcibly terminated with an error message.

```
/COMPILE EP=32
```

This record sets the page limit to 32 pages. Note that this estimate could be overridden in turn by a further run parameter specification on a /EXECUTE record.

When a page limit is given as a compiler parameter, it has no effect on the execution of a program run from an object deck.

## Run-Time Checking and Diagnostics

The Algol W run-time support system provides extensive checking for error conditions arising as a result of program execution. The following compiler parameters control how much checking is done and the behavior of the Algol W system when a fatal error is detected.

DEBUG | DEBUG=<number-of-levels> | DEBUG=ALL |
CHECK | NODEBUG | NOCHECK  Default: DEBUG=10

These parameters specify a hierarchy of three levels of run-time checking and diagnostic printing.

The default setting of DEBUG causes Algol W to generate code to:

(1) check at run-time for the following possible error conditions:

  (a) array subscripts out of range,
  (b) invalid substring index,
  (c) invalid case selection index,
  (d) incompatible reference field designator;

(2) retain sufficient information from the compiler identifier tables to allow a post mortem dump of active variables and their values to be printed if a fatal error occurs.

NODEBUG suppresses retention of the identifier tables.

NOCHECK suppresses both retention of the identifier tables and generation of code for run-time consistency checks. Please note the warning given later in this parameter description.

CHECK has no effect if DEBUG is in effect. If NOCHECK is in effect it will cause code for run-time consistency checks to be generated, that is CHECK overrides NOCHECK to provide the same midway compiler state as NODEBUG produces when overriding DEBUG.

The hierarchy is :

```
        DEBUG
     CHECK  NODEBUG
        NOCHECK
```

Going up this hierarchy increases the level of checking and diagnostic support available. It also increases the size of the compiled object code and slightly increases the compilation time. There is no difference in speed of execution for the top two levels of the hierarchy. However programs compiled with NOCHECK may run considerably faster, particularly if they contain many references to arrays.

Warning: While NOCHECK provides an increase in speed of execution, the considerable assumption is made that the program will always execute correctly for any data supplied to it. It is possible that some errors may not be detected as they occur and may give rise to more serious errors such as a program interrupt (59xx series error) elsewhere in the program. Worse still, the program may continue after failing to detect an error and thereby produce erroneous results. For this reason the Computing Center will not accept as evidence of a malfunction of the Algol W system the results of a program execution where the program was compiled with the NOCHECK parameter. It is for use strictly at the user's risk.

When a post mortem dump is produced as the result of a fatal error condition, the values of variables are dumped for the current block or procedure and then for the preceding blocks or procedures out to a predefined number of levels. Finally the outermost block is dumped if this has not already been done.

By default a post mortem dump will print the values of variables for 10 levels before skipping to the outermost block but this may be reset by an integer number given with the DEBUG parameter. This integer should be in the range 1 to 254.

If the keyword ALL is given, all levels will be dumped.

Example:

```
/COMPILE DEBUG=5
```

Note that the number of levels given with such a DEBUG parameter applies only to compile, load and go mode and may be overridden by a setting of the run-time DEBUG parameter.

ARRAYDUMP=<number-of-elements> Default: ARRAYDUMP=10

When arrays are dumped by the post mortem dumping mechanism, a predefined number of elements are dumped and then the value of the last element is printed, if this has not been done already. By default 10 elements will be printed but this may be reset by giving an integer value in the range 1 to 254 with the ARRAYDUMP parameter.

The keyword ALL will dump all elements.

Example:

```
/COMPILE ARRAYDUMP=1
```

This would print only the values of the first and last elements in any array dumped.

Note that the number of elements specified with such an ARRAYDUMP parameter applies only to compile, load and go mode and may be overridden by a setting of the run-time ARRAYDUMP parameter.

## Miscellaneous Parameters

The following parameter is ignored by Algol W but is documented here for completeness.

ID=<identifier>

<identifier> is a string of 1 to 8 alphanumeric characters.

The following parameters control the generation of object code.

SYNTAX | GENERATE Default: GENERATE

SYNTAX causes the compiler to cease execution after pass two. The program is checked for grammatical correctness only: no object code is generated, as this is done by the third and final pass in the compiler. SYNTAX provides a cheap way of checking the correctness of an Algol W program without using the additional resources required for code generation.

GENERATE, which is the default, is the opposite of SYNTAX. Grammatically correct programs will be compiled to object code.

The remaining parameters in this section are provided for those maintaining the Algol W compiler. While they may be of some interest to Computing Science students they are documented here only for completeness.

ALIST

This parameter will print a pseudo-assembly listing of any generated object code produced by the compiler. Instructions are identified by the usual System/370 mnemonics. Object code and the code segment location counter are printed in hexadecimal. The Algol W source program co-ordinate is printed, in decimal, to the left of this information.

TABLES

This parameter will print the contents of the compiler internal tables: in particular the compiler identifier name table, the block list and the segment table.

CTRACE

This parameter combines the effect of the two previous ones and in addition prints the edit code (tokenized form) from pass one and the tree(s) produced by pass two.

PTRACE

This parameter produces a trace of the pass two parser algorithm.

PSTACK

If this parameter is in effect and a syntax error is detected, the current contents of the parser stack will be printed with an error message.

## RUN-TIME PARAMETERS

Run-time parameters control the action of a compiled Algol W program during execution. Where they may appear depends on whether the program is executing under the control of the compile, load, and go system or is being run from an object deck.

When being run in compile, load, and go mode, a subset of the run-time parameters may be set using the relevant compiler parameters as supplied on a /COMPILE or /GLOBAL record or in in the PAR= field of the $RUN command which invoked *ALGOLW. This subset is:

ETIME | TIME
EPAGES | PAGES
DEBUG
ARRAYDUMP

When these parameters are given in the form of compiler parameters, they effectively reset the defaults for the run-time system for this program only.

Run-time parameters may be given in two other ways. They may be stored with the compiled program object using the RUNPARM compiler parameter to supply an initial run parameter string. They may also be supplied explicitly when program execution is started, in which case they are given on a /EXECUTE record in compile, load and go mode, or in the PAR= field of the MTS $RUN command which invokes an Algol W object deck stored in a file.

The order of evaluation of run-time parameter strings is as follows:

(1) Compiler parameters which provide run-time functions, that is those from the list above. This only applies in compile, load and go mode.

(2) The initial run-time parameter string supplied as the right-hand side expression of a RUNPARM compiler parameter.

(3) Parameters supplied explicitly when the program is run, either from a /EXECUTE record or a PAR= field on the RUN command.

Any or all of these parameter fields may, of course, be blank. No run-time parameters are mandatory. Note also that the DATAPARM run-time parameter described below has the property of suspending step (3) if given in step (2).

The parameters and their descriptions follow.

SIZE=<integer>[{K|P}] Default: SIZE=16P

The SIZE run-time parameter is available only when a program is being run from a file. It specifies the size of main storage which the run-time system is to acquire for use by the executing program for storage of all variables and records. If a program only occasionally requires storage larger than the default size, this parameter may be given when the program is run. On the other hand, if the program regularly requires more than the default storage, or will not run at all without an overriding parameter, then it is wise to include a SIZE specification via the compile time RUNPARM parameter.

SIZE is specified as an integer number of bytes followed by an optional scaling factor: K (kilobytes, value 1024) or P (pages, value 4096). The default allocation of 16P therefore allocates 65536 bytes.

In compile, load and go mode, working storage is provided at run-time by the release of storage provided for the compiler. The size of the run-time working storage is therefore decided in this case by the compile-time SIZE parameter.

FILE=<filename>

FILE can be used to specify a filename containing data records to be input to the program during execution. This parameter is similar, but not identical to, the FILE compiler parameter.

When FILE=<filename> is given, a user defined input/output stream named "FILE" is created during Algol W's run time initialization and assigned to the specified <filename>. The basic input stream is then set to be FILE instead of the default INPUT (MTS SCARDS). This means that any program which uses for input the predeclared procedures Read, Readon or Readcard, or the special basic input stream name Rdr, will read in from the filename specified in the FILE= assignment.

This behavior differs from the compiler FILE parameter because the records are not read via the INPUT stream. Control records are not therefore valid within the specified filename and will be treated as data records.

For example:

```
/EXECUTE FILE=W702:JDATA
```

Read statements within the program fetch data records from the file W702:JDATA, rather than the INPUT stream.

ETIME=<number>[{S|M}] | TIME=<number>[{S|M}] Default: see text

When an Algol W object program executes, a limit may be set on the total cpu time which may be used by the program. The time limit is determined as follows:

(1) If the program is executing in compile, load and go mode, and if a limit is set by the run-time ETIME parameter then this is used.

(2) If no run-time limit is subsequently set, but a compiler ETIME parameter specifies one, then this time is used. The run-time parameter overrides the compiler parameter.

(3) If no ETIME parameter of any kind is given, Algol W tries to find out if a system time limit exists, and, if so, when the program starts to execute the most immediate system limit is determined. This may be either a local time limit on the MTS $RUN command, or a global time limit on an MTS batch job. If none exists, no time limit is set.

(4) If a such limit does exist, an Algol W time limit is set which is slightly ahead of the system one. This allows Algol W to take control before the system does so.

Using the ETIME parameter, alternative values between 0.2 seconds and 60 minutes may be specified. They are given as an unsigned number followed by an optional scale factor. If no scale factor is given, or it is S, then the quantity is taken to be specified in seconds. If the scale factor is M, it is taken to be in minutes. The number given may be specified with up to three places of decimals.

ETIME and TIME are synonyms. They specify the execution time of a program. If this time estimate expires before the program completes execution, then execution is forcibly terminated and a fatal run error message printed. For example:

```
/EXECUTE ET=3.5S
```

This record sets the time estimate for the program to 3.5 seconds. Note that this estimate would override any previous compiler parameter, for example on a /COMPILE record.

EPAGES=<integer> | PAGES=<integer> Default: see text

When an Algol W object program executes, a limit may be set for the maximum number of pages of printed output which it may produce. The page limit is determined as follows:

(1) If a limit is set by the run-time EPAGES parameter then this is used.

(2) If no run-time limit is subsequently set, but a compiler EPAGES parameter specifies one, then this is used. The run-time parameter overrides the compiler parameter.

(3) If no EPAGES parameter of any kind is given, Algol W tries to find out if a system page limit exists, and, if so, then when the program starts to execute the most immediate system limit is determined. This may be either a local page limit on the MTS $RUN command, or a global page limit on an MTS batch job. If none exists, no page limit is set.

(4) If such a limit does exist, an Algol W page limit is set which is 1 page less then the system one. This allows Algol W to take control before the system does so.

Using the EPAGES parameter, an alternative value for the limit is specified as an unsigned integer between 1 and 200.

The page count is maintained with the following assumptions:

(1) Before any records are output, the printer is positioned on the perforations between two pages.

(2) Carriage control is in effect. All of the MTS logical carriage control characters are accepted.

(3) Printer pages are assumed to have a physical size of 66 lines, containing a 3 line header, a 60 line logical page and a 3 line trailer. Use of the ANSI standard logical carriage control characters (space, 0, -, 1) will write into the logical page only, skipping the 6 line trailer and header from the bottom of one logical page to the top of the next.

(4) Only output to the PRINT stream (MTS SPRINT) is counted, unless the executing program specifies otherwise using a Qualify predeclared procedure call with the PAGELIMIT parameter.

EPAGES and PAGES are synonyms. If an executing program attempts to print more pages than the estimate on the streams for which pagecount checking is enabled, program execution is forcibly terminated with an error message.

```
/EXECUTE EP=5
```

This record sets the page limit to 5 pages. Note that this estimate would override any compiler parameter specified, for example on a /COMPILE record.

CC | NOCC Default: CC

Where an executing Algol W program performs output operations using the predeclared procedures Write, Writeon and Writecard, carriage control characters are generated automatically in column one of each output record. The NOCC parameter suppresses this action so that the first character of each line is now the first character specified by the relevant output statement. This can be used as a method either of supplying an alternative carriage control character explicitly or of removing them when they are not required, that is when writing a data file.

CC overrides NOCC and will cause automatic carriage control character generation.

Note that the effect of these parameters can be achieved within a program by assignments to the predeclared logical variable Write_Cc which these parameters in fact set. Setting Write_Cc to 'false' is the equivalent of specifying NOCC as a run-time parameter.

DATAPARM

This is only effective if specified in the initial run time parameter string given as the right-hand side of a RUNPARM compiler parameter. Its effect is to suspend keyword processing of any subsequently encountered run-time parameter string. The latter, which might have come from either a /EXECUTE record or a $RUN PAR field is padded on the right with blanks to a length of 256 bytes and passed, in its entirety without editing, to the predeclared string(256) variable Sysparm. This enables a user to avoid Algol W's parameter processing but still examine the parameter field using the logic of the executing program.

DEBUG | DEBUG=<number-of-levels> |
DEBUG=ALL | NODEBUG Default: DEBUG=10

This parameter is only effective if the DEBUG compiler parameter is also in effect. Otherwise the object code will include insufficient information to perform the required action.

DEBUG without any expression implies DEBUG=10.

When a post mortem dump is produced as the result of a fatal error condition, the values of variables are dumped for the current block or procedure and then for the preceding blocks or procedures out to a predefined number of levels. Finally the outermost block is dumped if this has not already been done.

By default a post mortem dump will print the values of variables for 10 levels before skipping to the outermost block, but this may be reset by an integer number given with the DEBUG parameter. This integer should be in the range 1 to 254.

The keyword ALL will dump all levels.

Example:

```
/EXECUTE DEBUG=5
```

NODEBUG suppresses any post mortem dump should a fatal run error subsequently occur.

ARRAYDUMP=<number-of-elements> Default: ARRAYDUMP=10

When arrays are dumped by the post mortem dumping mechanism, a predefined number of elements are dumped, and then the value of the last element is printed, if this has not been done already. By default 10 elements will be printed, but this may be reset by giving an integer value in the range 1 to 254 with the ARRAYDUMP parameter.

The keyword ALL will dump all elements.

Example:

```
/EXECUTE ARRAYDUMP=1
```

This would print only the values of the first and last elements in any array dumped.

ALLTIMES | NOTIMES | TOTALCPU |
PROBLEMCPU | SUPERCPU | ELAPSED  Default: see text

These parameters override the default setting for the timing print which Algol W writes to SERCOM when a program ceases execution. Their action is as described under the equivalent parameters to the predeclared procedure Iocontrol. These are described in the Section "Timing Information" in the section "Miscellaneous Topics."

The default setting is TOTALCPU when running from a conversational terminal and ALLTIMES in batch.

<string>  Default: " "

If a run-time parameter is not identifiable as one of the above keywords or keyword expressions, it is examined to see if it is a character string delimited by either quotes (") or primes (’). If this is found to be the case, the value of the character string, after compressing double delimiters within the string to single characters, is padded on the right with blanks to a length of 256 bytes and placed in the predeclared string (256) variable Sysparm.

If more than one run-time parameter specifies a string, only the last string encountered will be processed and passed in Sysparm. No concatenation takes place.

## THE ALGOL W COMPILER SYSTEM

The Algol W compiler provides an implementation of the Algol W language. No such implementation can perfectly reflect the language design in all respects, so certain additional information is necessary to a programmer preparing code for this particular compiler. The following sections provide such implementation dependent information for the MTS compiler accessed via *ALGOLW.

## Symbol Representation

Algol W programs are built up from the following basic character set:

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9
" # ’ | , ; : . ( ) + - * / ¬ _ = < > %
```

These characters are used according to the rules of the language to form identifiers, reserved words, and special symbols. The reserved words of Algol W are:

```
abs   algol   and   array   assert   begin   bits   case
comment   div   do   else   false   for   fortran   go
goto   if   integer   is   logical   long   not   null
of   or   procedure   real   record   reference   rem
result   shl   short   shr   step   string   to   true
until   value   while
```

A full list of all Algol W basic symbols, with their meanings, is given in Appendix D. See also the section on "Input Format" later in this section for details of case independence.

Note that embedded blanks are not allowed in reserved words, identifiers and numbers. Adjacent reserved words, identifiers and numbers must be separated by at least one blank. Otherwise blanks may be used freely in an Algol W program.

## Predeclared Identifiers

The following identifiers are predeclared in the Algol W environment but may be redeclared by the user:

```
A_Count   Arccos   Arcsin   Arctan   Assign   Attnmark   Attntrap
Base10   Base16   Bitstring   Call   Canreply   Cmd   Code
Control   Cos   Cosh   Cot   Cxcos   Cxexp   Cxln   Cxsin
Cxsqrt   Date   Decode   Divzero   Empty   Endfile   Entier
Epsilon   Erf   Erfc   Error   Exception   Exp   Exponent
External   Fetch   Filemark   Flush   Fn_Value   Fullword
Function   Gamma   Get   Getcard   Geton   Getstring   Halfword
I_W   Imag   Imagpart   Input   Intbase10   Intbase16
Intdivzero   Intovfl   Iocontrol   Link   Ln   Lngamma   Locate
Log   Longarccos   Longarcsin   Longarctan   Longbase10
Longbase16   Longcos   Longcosh   Longcot   Longcxcos   Longcxexp
Longcxln   Longcxsin   Longcxsqrt   Longepsilon   Longerf
Longerfc   Longexp   Longgamma   Longimag   Longimagpart   Longln
Longlngamma   Longlog   Longrealpart   Longsin   Longsinh
Longsqrt   Longtan   Longtanh   Lowercase   Maxinteger   Maxreal
```

Move Newline Number Odd Ovfl Pi Print Protect
Punch Put Putcard Puton Putstring Qualify R_Cmplx
R_Code R_D R_Expchar R_Float R_Format R_Sig R_W
Rcall Rdr Read Readcard Reader Readon Realpart
Release Rewind Round Roundtoreal R0 R01 R1 S_W
Sense Sin Sinh Sqrt Stop Store Syscode Sysindex
Sysparm Tan Tanh Time Trace Translate Truncate Unfl
Uppercase User Write Write_Cc Writecard Writeon
Writer Wtr Xcpaction Xcplimit Xcpmark Xcpmsg Xcpnoted
Xdelete Xgetcard Xputcard

These identifiers are of three types:

(1) Predeclared procedures, which provide input/output and other facilities. See the list in Appendix E.

(2) Predeclared functions, which provide standard analytic functions and transfer between data of differing simple types. See the list in Appendix F.

(3) Predeclared variables, which are provided for a variety of different reasons. See the list in Appendix G.

## Restrictions

The implementation imposes the following restrictions which may affect some users:

(1) Identifiers consist of a maximum of 256 characters.

(2) A maximum of fifteen record classes may be declared.

(3) Approximately 256 constants are allowed in a procedure or a nontrivial block (including constants in trivial blocks contained in the procedure or nontrivial block).

(4) The maximum number of dimensions for an array is 15.

(5) A maximum of 34 arguments is allowed in a procedure call.

(6) Not more than 999 procedures or blocks containing declarations (that is nontrivial blocks) are allowed.

(7) The data area at run-time for each active procedure or nontrivial block, excluding array elements, is limited to 4096 bytes.

(8) The total amount of space occupied by the constants and machine code in any procedure or nontrivial block may not exceed 8192 bytes.

(9) No block may be included in more than 29 other blocks.

(10) Nontrivial blocks, blocks associated with procedures, argument lists, For statements and labels may not be nested within each other to a depth of more than eight counting the initial 'begin'. In other words, no declaration is allowed below the eighth block nesting level.

(11) A maximum of 256 external references generated by 'algol' or 'fortran' constructs is allowed per program or precompiled procedure.

(13) Not more than 63 procedures may be referenced within a single procedure. Here "procedure" means a procedure, nontrivial block or an external reference generated by the use of the Call or Rcall predeclared procedures or the External predeclared function.

(13) A maximum of one megabyte (1048576 bytes) of storage may be allocated to a program at run-time for all variable and record storage. This is a restriction imposed by the MTS operating system.

## Input Format

The compiler accepts input records of any length. However only the first 255 characters of an input line will be scanned for Algol W source text. Additionally, the Algol W compiler's input scanner routine regards each input line as being separated from the next by a single space. This implies that identifiers, reserved words and constants may not be broken across an input line boundary. For string constants, an extension of the syntax allows strings to be broken over a line boundary in a way best described by example:

```
S := "ABCDEFGHIJK"
     "LMNOPQRSTUV"
     "WXYZ";
```

This statement is exactly equivalent to:

```
S := "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
```

Note that this form of string concatenation applies to string constants only.

The implemenation of Algol W is case independent. Identifiers, reserved words and arithmetic or bits constants may be entered in any mixture of upper and lower case. When lower case alphabetics are encountered in an identifier or a reserved word, they are assumed to represent the equivalent uppercase character. Differences in case only

do not distinguish two otherwise identical identifiers or reserved words. String constants may use any of the 256 possible EBCDIC character encodings (see Appendix B) and, if alphabetic, the case of such text is preserved by the compiler. However the following instances of string use will work correctly with any mixture of upper and lower case.

(1) External references: using the Call or Rcall predeclared procedures, the External predeclared function or the 'algol' or 'fortran' constructs.

(2) Procedure references in the Link predeclared function.

(3) Values of the predeclared variable R_Format or format codes used in format directed input/output.

(4) Strings used as input/output stream designators.

(5) String keywords to the Control, Iocontrol, Qualify or Sense predeclared procedures.

External references and Control keywords are capitalized by the Algol W system before use.

## SYSTEM OUTPUT

The compiler and run-time system produce the following output:

(1) A compiler source listing is written to SPRINT if assigned, or to the file or device specified via an SLIST compiler parameter. This output may optionally include an identifier cross reference table.

(2) In production mode, an object deck is written to SPUNCH if assigned, or to the file or device specified via a DECK compiler parameter.

(3) Compiler diagnostic messages, including timing information, are written to SERCOM.

(4) Run-time diagnostic messages, including timing information, are also written to SERCOM (the ERROR stream).

The following MTS file listing shows a file containing a valid Algol W program and control records:

```
# $LIST qqsv
 >       1     /COMPILE DECK,SLIST,INDENT,TRIDENT=MC,TRRESERV=LC
 >       2     /TITLE "SUM OF SERIES"
 >       3     BEGIN
 >       3.5   INTEGER SUM_SQUARE, SUM_CUBE, LIMIT;
 >       4     READ(LIMIT);
 >       5     SUM_SQUARE := SUM_CUBE := 0;
 >       6     FOR INDEX := 1 UNTIL LIMIT DO
 >       7     BEGIN  INTEGER SQUARE, CUBE;
 >       8     SQUARE := INDEX * INDEX;
 >       9     SUM_SQUARE := SUM_SQUARE + SQUARE;
 >      10     CUBE := SQUARE * INDEX;
 >      11     SUM_CUBE := SUM_CUBE + CUBE
 >      12     END FOR_LOOP;
 >      13     WRITE("SUM OF SQUARES = ", SUM_SQUARE);
 >      14     WRITE("SUM OF CUBES   = ", SUM_CUBE)
 >      15     END OF_PROGRAM.
 >  END OF FILE
```

As explained earlier in this section, it can be compiled using *ALGOLW in the following way:

```
# $RUN *ALGOLW SCARDS=qqsv
# EXECUTION BEGINS
  Source listing:  "-AWLIST"
  Object program:  "-AWLOAD"
  (MAIN)    0.14 seconds to compile,  size 824 bytes
# EXECUTION TERMINATED
```

The messages lines shown above are diagnostic information, written to SERCOM. Note that the CPU time given is the total time, including system overheads for input/output performed by the compiler.

The following subsections describe the compiler output on the various input/output streams, mostly by reference to the example program and compilation above.

## Source Program Listing

The listing file from the program given at the head of this section will look like this:

```
Algol W : File=W701:QQSV                    SUM OF SERIES

      3     0000 1-     begin
      3.5   0001 --       integer Sum_Square, Sum_Cube, Limit;
      4     0002 --       Read(Limit);
      5     0003 --       Sum_Square := Sum_Cube := 0;
      6     0004 --       for Index := 1 until Limit do
      7     0004 2-       begin  integer Square, Cube;
      8     0006 --         Square := Index * Index;
      9     0007 --         Sum_Square := Sum_Square + Square;
     10     0008 --         Cube := Square * Index;
     11     0009 --         Sum_Cube := Sum_Cube + Cube
     12     0009 -2       end For_Loop;
     13     0010 --       Write("SUM OF SQUARES = ", Sum_Square);
     14     0011 --       Write("SUM OF CUBES   = ", Sum_Cube)
     15     0011 -1     end Of_Program.

Options (UN230) :- main, debug

(MAIN)    0.14 seconds to compile,  size 824 bytes
```

Some variation in the form of the compiler source listing is possible by specifying the appropriate compiler parameters.

The first line on each listing page gives the source input filename and any title string supplied on a /TITLE control record. It also gives the time, date and listing page number. This last information is on the extreme right of the listing line and hence does not appear in the example above.

Taking the listing columns from left to right:

(1) Source file line number. The MTS line number from which the source input record was taken is printed in this column. If the input record echoed on this listing line is a continuation, the line number field is blank and the most immediate previous line number applies. This field is not printed when the compiler parameter NONUMBER is in effect.

(2) Source statement number. This field contains the statement (or co-ordinate) number referenced in all Algol W error messages. It is a count from zero of each 'begin' and each semicolon (;), excluding those ending comments.

(3) Block nesting level indicator. This two character code indicates the 'begin'...'end' block nesting depth. Normally both characters are "-", but when either 'begin' or 'end' appears on a source statement the change in block level is indicated in the characters of this indicator. Each 'begin' encountered increments the block nesting counter by one and the first character of the indicator shows the level after the 'begin'. After each 'end' the block nesting counter is decremented by one and the last character of the indicator shows the level after the 'end'.

(4) Source record image. The remainder of the listing line shows the source input record itself. Note that both the production and form of the source record image are under the control of many compiler parameters and control records as described earlier in this section. In this example the three parameters INDENT, TRINDENT=MC and TRRESERV=LC, among others, were included on the /COMPILE card. Thus in this listing the source images are indented three spaces per block level, the indentifiers are translated to a mixed case form, and the reserved words are translated to lower case.

At the end of the listing two further lines appear:

(1) The "Options" line gives a list of the most important compiler parameters in force, mainly those which affect subsequent execution of the object program produced. The five characters within parentheses, here "UN230", are an encoded release date for this particular version of the Algol W system.

(2) The second line shown is the last listing record output by the compiler for each compilation. The first item in it is either "(MAIN)" for a main program or the external symbol definition name (ESDname) for a precompiled procedure. The rest of the line gives the total CPU time used by the compiler during compilation, and the total number of bytes of code generated in the resultant object program.

## Compiler Diagnostic Output

In the previous example the program compiled successfully. Where the rules of the Algol W language are violated, the compiler will issue an error message (or messages) on the ERROR stream, MTS SERCOM. If the listing device and the ERROR stream do not have the same assignment, the compiler diagnostic messages are also written to the listing file, in which case they appear after the source listing (and cross reference, if any) but before the final summary line. If the messages indicate fatal errors preventing successful compilation of the program, the final listing line will indicate this with the words "no code generated."

A list of all compiler (and run-time) error messages, with full explanations, is given in Appendix C.

When the Algol W compiler is run at a conversational terminal, the ECHO parameter is the default and the relevant part of the source program listing is printed preceding the error message. If in the program used in the previous example of a source listing, the variable Square on line 8 had been misspelled as Squre, the following output would have appeared:

```
# EXECUTION BEGINS
  Source listing:  "-AWLIST"

       8     0006  Squre := Index * Index;

  Error 2002 near 0006 - "squre" is undeclared
     (Found near " ; squre ")

  (MAIN)    0.17 seconds to compile,  no code generated
# EXECUTION TERMINATED  RC=16
```

This output appears on the terminal because the MTS SERCOM device name defaults to the "master sink," which is the terminal when running conversationally and the line printer in batch.

Because the listing is being sent to a different destination, a copy of these diagnostic messages will also appear in the listing file "-AWLIST".

The Algol W compiler may produce more than one message for each actual coding error detected. If this is the case, and if some of the error messages are not clear, the technique recommended is to fix as many errors as possible and then recompile. If actual errors remain in the program the new set of messages should be more concise and may well provide clearer information.

## Identifier Cross Reference Listing

Specification of the XREF compiler parameter will cause a cross reference listing to be appended to the source program listing. For the program used in the above examples this would take the form:

```
Algol W : File=W701:QQSV                      Cross Reference

    27 references

Cube           0005  0008  0009
Index          0004  0006  0006  0008
Limit          0001  0002  0004
Read           0002
Square         0005  0006  0007  0008
Sum_Cube       0001  0003  0009  0009  0011
Sum_Square     0001  0003  0007  0007  0010
Write          0010  0011
```

Cross reference listings are very useful for much larger programs than can be shown in examples in this manual.

The listing is output in two columns to optimize paper use. The identifier names are listed alphabetically, with all the source state-

ment numbers where each is used printed after the name, eight to a line. A maximum of 14 characters of the identifier name will be printed; use of the same name in declarations of different identifiers is not distinguished in the cross reference listing.

If insufficient storage is available for cross reference processing, a partial cross reference will be output with a warning message at the head. Running the compiler with a larger SIZE parameter will produce the full cross reference.

## Object Deck Output

When an Algol W program is successfully compiled, an object deck is produced.

When running the compiler in compile load and go mode, this object deck is immediately passed to the Algol W loader without any action being required on the part of the user.

In production mode, the object deck is output to a file as specified either through a DECK compiler parameter, or an assignment to the SPUNCH logical device name on the MTS $RUN command.

The object deck so output contains one object module (representing one control section) for each program segment compiled. A program segment is a program, a procedure or a nontrivial block (one containing variable declarations). Each module contains the standard object cards (ESD, TXT, RLD, END) as described in MTS Volume 5, System Services.

For a main program only, the first two modules output do not correspond to any program segment. They are:

(1) Module name : AWXSTART. This contains lead in information used by the run-time system to locate the relevant modules at Algol W system initialization. Algol W object programs always enter at AWXSTART.

(2) Module name: AWXRCTBL. This contains information about record declarations and is used by the run-time system to keep track of records created during program execution.

For a main program, module names are of the form:

```
AWXSCnnn
```

where "nnn" is a three character group running 001, 002, 003, etc. AWXSC001 is the first segment of an Algol W program proper; the run-time initialization routines called by AWXSTART in turn call AWXSC001 to start program execution.

For a precompiled procedure, the module name of the first segment is always the procedure name capitalized and truncated to eight characters if necessary. If the procedure is complex enough to warrant more than one segment, subsequent object module names are formed by truncating the name further to five characters (or padding with hash marks (#) to that length), and appending 002, 003, 004, etc. to form an eight character ESDname.

Specification of a compiler parameter such as:

```
LIBSEARCH=<filename>
```

will cause the compiler to output a record:

```
$CONTINUE WITH <filename> RETURN
```

at the end of the normal object deck output. The MTS loader will load from the specified object or library filename when the program is $RUN.

## Run-Time Diagnostics

Programs which compile successfully do not always run correctly. If the program used in the previous examples is run with the data record:

```
10
```

then the program will function correctly producing the output:

```
SUM OF SQUARES =            385
SUM OF CUBES   =           3025

0.02 seconds in execution
```

If on the other hand the data record was:

```
400
```

then trouble is in store when the program is run. The sum of cubes computed in this program will exceed the maximum integer which can be represented on the machine. The following diagnostic output will be produced:

```
Run error 5908 near 0009 in (MAIN) -- Integer overflow exception.

              Algol W Post Mortem Dump
Trace of active segments
Dump of local variables near coord 0009 in <BLOCK>
   Square = 92416                  Cube = 28094464

Dump of local variables near coord 0004 in (MAIN)
   Sum_Square = 9411080            Sum_Cube = 2121155136
   Limit = 400     Index = 304

0.03 seconds in execution
```

This is written to the ERROR stream, MTS SERCOM, so that it would be seen even if the normal output was being sent to a file.

The first line of the diagnostic output is the run error message giving the cause of the failure, in this case integer overflow. This message may run to several lines depending on the cause of the error. A list of run time error messages, with full explanations, is given in Appendix C.

The remainder of the diagnostic output gives the values of the program variables at the time of the error. This is called a post mortem dump, for fairly obvious reasons. The dump can only appear if sufficient information has been retained when the object program was produced by the compiler. This is under the control of the DEBUG parameter; it is the default, so a post mortem dump is normally produced when a run-time error occurs.

Note the order in which the variables are dumped. The system starts with those nearest the point of error, and then dumps segments outwards until the main program block is reached.

By default the post mortem dump routines will dump ten program segments and then skip to the outer (main program) block, which it will dump before terminating. This may dump all active blocks; alternatively a message such as:

```
...  58 segments skipped
```

may appear, indicating that, in this case, 58 active segments have been skipped. A large number of segments may have been active through an involved series of procedure calls or because a procedure has called itself recursively many times. The number of segments dumped is under the control of the DEBUG compiler and run-time parameters.

When the values of array elements are dumped, a similarly abbreviated form of print is the default action. The first ten elements and the last are dumped. For a small array this may dump the whole array; for a larger one a dump such as:

```
A(1) = 84       A(2) = 91       A(3) = 14       A(4) = -76
A(5) = -96      A(6) = -28      A(7) = 66       A(8) = 99
A(9) = 41       A(10) = -54     ...             A(4000) = -68
```

might appear. The elipsis (...) indicates that elements have been skipped in the dump of A, a 4000 element single dimension array. The number of array elements dumped is under the control of the ARRAYDUMP compiler and run-time parameters.

Reference variables dump as an occurrence of the record class of which they are a member. The occurrence is an integer constant linked by a period (.) to the record class name. For instance:

```
Theatre = 25.Main_List
```

indicates that the reference variable Theatre points to the 25th instance of a record of class Main_List.

The values of variables which have not been assigned are displayed in the dump as a question mark (?). In the case of string variables a repeated pattern of question marks indicates the part of the string which has not been assigned a value.

## System Return Codes

The Algol W system leaves a return code in general register fifteen on exit. This may be tested by MTS conditional commands and appropriate action taken.

The compiler return codes are:

- 0 Program compiled correctly.
- 12 Program compiled and the object program is loadable, but nonfatal pass three errors have marked certain paths through the code as nonexecutable.
- 16 Fatal errors were detected during compilation. Either no or an incomplete object program was produced.

The run-time system return codes are:

- 0 Program execution terminated normally.
- 4 Program termination was caused by an end-of-file condition processed using the Endfile predeclared reference.
- 8 Program termination was caused by a fatal run error other than end-of-file. When an Algol W object program is run from a file, it will terminate with one of these run-time codes.

When *ALGOLW is run, either in compile, load, and go mode, or production mode, the single return code on system termination is the maximum of the individual step return codes for each of the compilation and execution phases performed.

# APPENDIX A: AN ALGOL W BIBLIOGRAPHY

This manual is the definitive document describing the Algol W language and its current implementation. However, the following bibliography lists material that is concerned with either Algol W or the design of Algol-like languages in general, and may be of interest.

Wirth, Niklaus and Hoare, C.A.R., A Contribution to the Development of Algol, Communications of the A.C.M., 9:6, June 1966

This paper contains the original published description of the Algol W language, although it differs in several respects from actual implementations.

Bauer, H., Becker, S. and Graham, S., Algol W Implementation, Stanford University Technical Report CS98, Stanford, Calif., USA, 1968

The original internal documentation for the Stanford implementation. It is now considerably out of date in many respects.

Genuys, F. (Editor), Programming Languages, Academic Press, London, England and New York, N.Y., 1968

This book is a collection of the text of a series of lectures given at a summer school in Villard-de-Lans, France, in 1965. The Algol W interest is in the section by C.A.R. Hoare on "Record Handling".

Bauer, Henry R., Introduction to Algol W Programming, Computer Science Department, Stanford University, Stanford, Calif., 1970

The original Stanford tutorial document on Algol W.

Aho, Alfred V. and Ullman, Jeffrey D. Principles of Compiler Design, Addison-Wesley, Reading, Mass., 1978

For any student of compiling techniques, this popular volume is one of the standard works.

Sites, Richard L., Algol W Reference Manual, Computer Science Department, Stanford University Technical Report STAN-CS-71-230, Stanford University, Stanford, California, 1972

The original Algol W reference manual, now replaced in MTS by this present manual.

Kieburtz, Richard B., Structured Programming and Problem Solving With Algol W, Prentice-Hall, Englewood Cliffs, N.J., 1975

Satterthwaite, Edwin H., Source Language Debugging Tools, Stanford University Technical Report STAN-CS-75-494, Stanford, Calif., USA, 1975

Another valuable source of Algol W internals documentation. Ed Satterthwaite worked on the development of Algol W during the period 1969-72.

Marsland, T.A. (Editor), Algol W References, Technical Report TR75-15, University of Alberta Computer Science Dept., U. of A., Edmonton, Alta, Canada, 1975

Contains a description of the original Alberta/Manitoba format directed input/output extensions, on which the present scheme was based.

Palay, Roger M. and Benson, Peter, FANGET AN: An Algol W Primer,Collegiate Publishing, Columbus, Ohio, 1978

An excellent text for use as an introduction to computer programming. The basic concepts of computing as well as Algol W are presented.

Palay, Roger M. and Benson, Peter, The Programming Psalter, Collegiate Publishing, Columbus, Ohio, 1978

A book of Algol W programming examples for use with "Fanget An".

Hunter, Alan and Hindmarsh, Margaret M., Algol W Development at Newcastle, Newcastle University Computing Laboratory, Technical Report Series, No. 124, 1978

The initial design specification for the present release of Algol W. Now inaccurate in several respects, as the actual implementation evolved from a discussion of this proposal amongst the member installations of the MTS Community.

# APPENDIX B: CHARACTER ENCODINGS

The following table presents the correspondence between printable string characters and their (EBCDIC) encodings. This encoding establishes the ordering relation on characters and thus on strings. Those characters in parentheses are available only on line printers having a TN print train, or on other printers with an equivalent character set. Integer codes not listed below do not correspond to any established character. (see Code, Decode in the section "Strings").

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 64 | space | 129 | (a) | 193 | A | 240 | 0 |
| 74 | (¢) | 130 | (b) | 194 | B | 241 | 1 |
| 75 | . | 131 | (c) | 195 | C | 242 | 2 |
| 76 | < | 132 | (d) | 196 | D | 243 | 3 |
| 77 | ( | 133 | (e) | 197 | E | 244 | 4 |
| 78 | + | 134 | (f) | 198 | F | 245 | 5 |
| 79 | \| | 135 | (g) | 199 | G | 246 | 6 |
| 80 | & | 136 | (h) | 200 | H | 247 | 7 |
| 90 | (!) | 137 | (i) | 201 | I | 248 | 8 |
| 91 | $ | 145 | (j) | 209 | J | 249 | 9 |
| 92 | * | 146 | (k) | 210 | K | | |
| 93 | ) | 147 | (l) | 211 | L | | |
| 94 | ; | 148 | (m) | 212 | M | 139 | ({) |
| 95 | ¬ | 149 | (n) | 213 | N | 155 | (}) |
| 96 | - | 150 | (o) | 214 | O | 173 | ([) |
| 97 | / | 151 | (p) | 215 | P | 189 | (]) |
| 107 | , | 152 | (q) | 216 | Q | | |
| 108 | % | 153 | (r) | 217 | R | | |
| 109 | _ | 162 | (s) | 226 | S | | |
| 110 | > | 163 | (t) | 227 | T | | |
| 111 | ? | 164 | (u) | 228 | U | | |
| 122 | : | 165 | (v) | 229 | V | | |
| 123 | # | 166 | (w) | 230 | W | | |
| 124 | @ | 167 | (x) | 231 | X | | |
| 125 | ’ | 168 | (y) | 232 | Y | | |
| 126 | = | 169 | (z) | 233 | Z | | |
| 127 | " | | | | | | |

Note particularly that the code sequences for the characters of the alphabet are in three parts, for "A"-"I", "J"-"R", and "S"-"Z", and that the upper case letters are 64 character positions after the equivalent lower case ones.

A full list of the 256 internal codes possible, together with their meanings, will be found in the "IBM System/370 Reference Summary", form number GX20-1850-3.

# APPENDIX C: ERROR MESSAGES

Error messages generated by the compiler, loader or run time system are listed here.

There are two basic types of errors: errors which occur during compilation and errors which occur while a compiled program is executing. Compilation errors are discussed below in subdivisions of pass 1, pass 2, and pass 3 error messages. Errors which occur while a compiled program is executing are called run-time errors. The messages they generate are the last type discussed in this appendix.

The compiler is divided into three passes: pass 1 reads the program, lists it, and saves it in memory in a compressed (tokenized) form; pass 2 parses the program, examining each statement to see if it is correctly formed; pass 3 generates the machine code for the program. Each pass is capable of detecting a different set of errors. (Pass 4, the loader pass, on rare occasions may also generate messages.)

All error messages from passes 1, 2, and 3 are of the form:

```
Error zxxx near yyyy - message
```

or:

```
Warning zxxx near yyyy - message
```

where "zxxx" is the error number, "z" is 1, 2, or 3 according to the pass which generates the message and "yyyy" corresponds to one of the coordinate numbers in the second column on the program listing. If there are several statements on a line, only the coordinate of the first one appears on the program listing. If any pass one or pass two error messages occur (other than warnings), then compilation stops at the end of pass two. Warnings do not cause compilation to stop.

Note however that if the compiler parameter TERSE has been specified, only one error per coordinate is reported, and any errors detected during pass one will stop compilation at the end of that pass.

PASS ONE ERROR MESSAGES

Error 1001 - incorrectly formed declaration

(1) 'string(x)' or 'bits(x)', where x is not a number.

(2) 'string(0)' or 'string(>256)'.

(3) 'bits(not 32)'.

Error 1002 - incorrect constant

(1) More than 256 digits.

(2) A bad exponent.

Error 1003 - missing "end"

A final "." or a control record encountered before an 'end' matching each 'begin'. (Check the block numbers in program listing.)

Error 1004 - unmatched "end" deleted

An 'end' encountered after what appeared to be the final 'end'. (Check the block numbers in the second column of the program listing.)

Error 1005 - missing ")"

'string(x' or 'bits(x' with no closing ")".

Error 1006 - illegal character

An erroneously punched or overpunched character. Overpunched characters may print as blanks; the card should be inspected in this case.

Warning 1007 - missing final "."

A control record encountered without a preceding ".". The compiler inserts a "." automatically.

Error 1008 - invalid string length

A string constant of length > 256, or a completely empty string.

An example of source program:

```
begin
   string(256) B;
   B := "this is a very very very very very very very very very"
        "very very very very very very very very very very very"
        "very very very very very very very very very very very"
        "very very very very very very very very very very very"
        "very very very very very very very very very very very"
        "very long string constant";
end.
```

The string constant assigned to B is too long.

Error 1009 - invalid bits length

(1) "#" not followed by hex digits.

(2) "#" followed by more than 8 hex digits.

Error 1010 - missing "("

'reference' not followed by "(".

Error 1011 - error table overflow

More than 50 error messages from pass 1. Subsequent errors are not listed.

Error 1012 - compiler table overflow

The program is too big to fit in memory during compilation. There is no more room in one of the tables constructed by the compiler. Recompile with a larger SIZE parameter to make the tables larger.

Error 1013 - identifier length > 256

An identifier is too long.

Error 1014 - unexpected "."

An apparently final "." not followed by a control record, such as in a constant with an inadvertently inserted space, as in this line: X := 3. 14

Error 1015 - too many record classes

Only 15 are allowed.

Warning 1016 - "else" preceded by (deleted) ";"

The sequence '; else' has been replaced by 'else'. Occasionally, this warning may appear along with a seemingly contradictory error message, namely,

```
   Error 2047 - expecting ";" but "else" was found
      (found  near "...")
```

This will happen if the following sample program is run:

```
   begin
      integer A, B, C;
      Read(A);
      if A < 0
      then
         B := 1;
         C := 2;
      else
         B := 0
   end.
```

The error in the above program is that the 'begin' and 'end' enclosing the block of the 'then' clause are omitted. The compiler first deletes the semicolon preceding the 'else', and then interprets the "B := 1;" as the end of the If statement and searches for a semicolon after the "C := 2". The messages returned are:

```
   Warning 1016 - "else" preceded by (deleted) ";"
   Error 2047 - expecting ";" but "else" was found
      (found near " 2 else ")
```

The problem can be corrected by enclosing the 'then' clause with a 'begin' and 'end', and removing the semicolon after the 2.

Error 1017 - too many blocks

A block is enclosed in more than 29 other blocks.

Error 1018 - compiler interface error

An error has been detected by the routines which fetch input records for the compiler, probably while processing a /COPY record. Details of the error appear within the compiler source listing.

Error 1019 - procedure name is reserved word

This error trap provides a clear error message for the case where a reserved word has been specified as a procedure name. Reserved words may not be used as identifiers. This error is commonly made by beginning Algol W programmers who specify something like:

```
   procedure result(..... );
```

'result' is a reserved word.

Error 1020 - unclosed string found at end of line

A common error made by Algol W users is to forget the closing quote (") for a string constant. This could lead to many errors being generated for the remainder of the program. To circumvent this problem, Algol W will not allow string constants to cross a line boundary. If an unclosed string is found the compiler produces the above error message and proceeds as if the string had been closed; that is it inserts a quote. For example:

```
Write("Results are as follows);
Write("A =", A, "B =", B, "C =",C);
```

A quote has been omitted after the word 'follows'. Algol W notices the error and closes the string by adding a quote to the end of that line. The program will still be in error but the misleading effects which could have arisen, such as parsing 'A =' as program text, will be avoided.

## PASS TWO ERROR MESSAGES

All pass 2 error messages are supplemented by:

(found near "...")

where "..." indicates a pair of symbols. In general, the first symbol is the input symbol or phrase after which the error was detected; the second is the next symbol to be scanned.

If any pass one or pass two error messages occur (other than warnings or notes), then compilation stops at the end of pass two. Several error messages may be generated for what is essentially a single mistake.

Error 2001 - more than one declaration of "<name>" in this block

The identifier <name> has been declared more than once in the same block.

Error 2002 - "<name>" is undeclared

The identifier <name> has not been declared in the current block or in one containing it.

Error 2003 - compiler error

The compiler has malfunctioned in some way. Leave the card deck or source file exactly as it is and present it to the appropriate member of the Computing Center staff.

Error 2004 - invalid procedure designator

In a call of the Link predeclared function, the parameter specifying the procedure to be called is invalid. It must be a string constant giving the name of the procedure only, with no leading, trailing, or embedded blanks.

Error 2005 - mismatched parameter

An actual parameter (that is an argument) in a procedure statement is not of a type compatible with the corresponding formal parameter in the procedure declaration.

Error 2006 - incorrect number of actual parameters

The number of actual parameters (that is arguments) in a procedure call does not equal the number of formal parameters in the procedure declaration.

Error 2007 - incorrect dimension

(1) The number of dimensions of an actual parameter (that is argument) array does not equal the number of dimensions declared for the corresponding formal parameter array.

(2) The wrong number of subscripts have been used in an array element reference.

Error 2008 - data area exceeded

The data space for each procedure or block with declarations is limited to 4096 bytes. Read the suggestions for error 3001.

Error 2009 - incorrect number of fields

In creating a record, too many or too few initial values have been specified.

Error 2010 - incompatible string lengths

(1) In String1 := String2, String2 is longer than String1.

(2) In String3(x|y), the substring specified exceeds the declared size of String3.

(3) A long string has been passed to a shorter formal string parameter. This could happen if the formal parameter was call by name or call by value.

(4) A shorter string has been passed to a longer formal string parameter. This could happen if the formal parameter was call by name, call by result, or call by value result.

Error 2011 - incompatible references

A reference variable refers to a record class to which it is not bound.

Error 2012 - blocks nested too deeply

Nontrivial blocks (that is blocks with declarations, or the blocks associated with a procedure) or actual parameter (that is argument) lists are nested more than eight deep. The error is detected early in the ninth block.

Warning 2013 - ";" should not follow expression

In 'begin ...<expression>; end' the semicolon is incorrect but ignored.

Error 2014 - reference must refer to record class

In 'reference (<name>)...' , <name> is not a record class.

Error 2015 - expression missing in procedure body

A function procedure must have its final value specified by an expression standing alone immediately before the 'end'.

Error 2016 - improper combination of types

Mixing incompatible types as alternatives of a conditional or Case expression.

Error 2017 - result parameter must be a variable

In a procedure declaration, a formal parameter is declared '... result <name>', but a call to that procedure has passed an expression which is not a variable.

Error 2018 - proper procedure ends with an expression

A procedure which returns no value nonetheless ends with an expression. (This will happen if a final assignment statement is using "=", instead of ":=".)

Error 2019 - syntax error in <item>

The program violates the Algol W syntax rules for the construction <item>. This message usually occurs where there are many legal constructions possible, so the compiler cannot be more specific. The compiler will recover by throwing away the whole statement, declaration, etc., in which the error occurs.

Error 2020 - array id used incorrectly

A simple variable must be used here.

Error 2021 - too many constants in procedure

Only 256 different constants (approximately) are allowed.

Error 2022 - incorrect string length

In S(x|y), y is less than or equal to zero, or greater than 256.

Error 2023 - compiler table overflow

The program is too big to fit into memory during compilation - there is no more room for the parse trees that represent the program. Recompile with a larger SIZE parameter or compile some procedures separately.

Error 2024 - too many procedures

Only 999 different procedures or blocks with declarations('code segments') are allowed by the compiler.

Error 2025 - constant out of range

(1) The absolute value of an integer is greater than (2**31)-1 (that is more than 9 digits).

(2) The absolute value of the adjusted exponent in a real number is greater than 75. (The exponent entered is adjusted to include the number of digits written in front of the decimal point in the mantissa.)

Error 2026 - index of array or string must be integer

(1) In S(x|y), x is not an expression of integer type.

(2) An array subscript is not an expression of integer type.

Error 2027 - incorrect operand type for <item>

<item> is a unary operator.

(1) 'long' is applied to something which is already 'long', or to string, bits, logical, or reference.

(2) 'short' is applied to something which is neither long real nor long complex.

(3) ¬ or 'not' (both mean not) is applied to something which is neither logical nor bits.

(4) Prefix + or - is applied to something which is logical, string, bits, or reference.

(5) 'abs' is applied to something which is logical, string, bits, or reference.

(6) In field Name(x), x is not a reference.

(7) In 'for I := x...', x is not an integer expression.

(8) In various other contexts, an integer or logical operand is required.

Error 2028 - incorrect operand type(s) for <item>

<item> is a binary operator. Even when the error is in the first operand, the error is detected after both operands are inspected.

(1) 'and' or 'or' is applied to expressions which are not both bits or both logical.

(2) A relational operator (such as > ) is applied to something which is complex, logical, or reference.

(3) 'shl' or 'shr' is applied to something which is not bits, or is followed by either an expression not enclosed in parentheses or a value which is not of integer type.

(4) In 'x is <recordclass>' , x is not of type reference.

(5) In 'x**y' , y is not of type integer.

(6) In a For statement, the 'until' expression is not of type integer.

(7) In various other contexts, an integer type operand is required.

Error 2030 - assignment incompatibility

An attempt to assign an expression of one type to a variable of a different type (or pass an actual parameter (that is argument) to a formal parameter of a different type). The only automatic conversions allowed are integer to real, integer to long real, real to/from long real, integer/real/long real to complex/long complex, complex to/from long complex (real cannot be assigned to integer without using Truncate, Entier, or Round).

Warning 2031 - name parameter; prefer "value result"
-- or --
Warning 2031 - at least one name parameter found

This is not an error, just a suggestion. In procedure declarations, it is more often the case that formal parameters have 'value' (and/or 'result') specified. Name parameters are very inefficient, and occasionally cause strange side effects. Check that leaving out 'value' (and/or 'result') is necessary. Very few algorithms require name parameters.

If the compiler parameter TERSE is in effect, the second form of the message is printed for the first name parameter only; other name parameters (if any) are passed without warning.

Error 2032 - simple variable id used incorrectly

The identifier in a substring designator is not type string.

Error 2033 - ... further messages suppressed

More than 64 errors detected; compilation continues with further messages suppressed.

Error 2034 - <xxx> found where <yyy> should be

The compiler expects to find <yyy> here, but the program actually has <xxx> instead. The compiler will recover by substituting some arbitrary <yyy> for <xxx>.

Warning 2034 - <xxx> found where <yyy> should be

The compiler expects to find <yyy> here, but the program actually has <xxx> instead. The compiler will continue as if <yyy> were really there instead of <xxx>.

Error 2035 - missing <xxx> (inserted)

The compiler expects to find <xxx> here, and guesses that it has been omitted. It will continue by inserting some arbitrary <xxx>.

Warning 2035 - missing <xxx> (inserted)

The compiler expects to find <xxx> here, and guesses that it has been omitted. It will continue as if <xxx> were really there.

Warning 2036 - <xxx> is illegal here (deleted)

This symbol (usually some punctuation symbol such as "," or ";") is not needed here - in fact, it is illegal here. The compiler will ignore it.

Warning 2037 - "result value" should be "value result"

According to the specifications of Algol W, 'value result' is legal but 'result value' is not. However, this compiler can understand either order, and should compile the program properly regardless.

Error 2038 - redundant or contradictory attributes

An example of redundant attributes in a declaration:

```
integer value integer result X
```

An example of contradictory attributes:

```
integer string Y
```

Error 2039 - a "procedure" parameter cannot be value, result, or array

A procedure parameter can only be specified as call by name. (A procedure parameter is the name of one procedure which is passed to another procedure as a parameter.)

Error 2040 - an array parameter cannot be value or result

An array can only be passed to a procedure by name; it cannot be a value or result parameter.

Warning 2041 - assignment operator should be ":="

An assignment statement was apparently intended, but "=" was employed instead of ":=" . Examples:

```
A := B
```

takes the value of B and puts it in A, and is called an assignment.

```
A = B
```

is a logical expression which compares A and B; its value is either 'true' or 'false'.

If a logical expression was intended, this message means that an assignment statement would have been legal here, but a logical expression is not. This message also appears if "=" is used instead of ":=" in a For statement; 'for I := ...' is the correct form.

Warning 2042 - true part of "if" not a simple statement

In an 'if ... then ... else' statement, the statement between 'then' and 'else' must be a simple statement. This restriction is necessary to avoid possible ambiguities.

Error 2043 - illegal quantity to left of ":="

In an assignment statement, the part to the left of the ":=" is something whose value cannot legally be changed in this way. Examples:

```
A + B := 5
```

is illegal because A + B is an expression, not a variable.

```
5 := A + C
```

is illegal because 5 is a constant, not a variable.

```
for A+B := 1 until N
```

is illegal because A+B is an expression, not a variable.

The left-hand side of the assignment operator := must be a variable, not a constant or an expression.

Warning 2044 - non-simple statement as predeclared procedure argument

Any statements among the parameters of a Write statement must be simple statements. The only reason for allowing statements here is to allow one to change the values of the output editing variables. If this is not the intention, this feature should not be used.

Error 2045 - compiler stack overflow

This message occurs when the program is too complicated for the compiler to handle. This usually means a very complicated expression with many levels of parentheses. Remedy: Try to rewrite the program to avoid such complicated constructions.

Error 2046 - declaration is not within a program

This declaration is not inside any main program. (Programs must begin with 'begin' and end with 'end.'.)

Error 2047 - expecting <xxx>, but <yyy> was found

The compiler is looking for some <xxx> here, but the symbol in the program is <yyy>, and <yyy> is not <xxx> (or is not a legal first symbol for an <xxx>). The compiler will recover by throwing away the whole statement, declaration, etc., in which the error occurs.

Error 2048 - type must precede other attributes

A parameter declaration must begin with the parameter's type (such as 'integer' or 'string'). Examples:

```
value integer X
```

is incorrect.

```
integer value X
```

is correct.

Error 2049 - no type specified in declaration

In the declaration of a parameter, the type has been omitted. Examples:

```
procedure A(X, Y);
real procedure B(value result Z);
```

In each example, a type (such as 'integer' or 'string') must be included for each parameter.

Error 2050 - expression found in middle of block

An expression has been found where a statement should be, among the statements of a block. The error is not detected until the end of the expression. This can be caused by writing "=" instead of ":=" for an assignment.

Warning 2051 - goto statement; prefer other control structures
-- or --
Warning 2051 - at least one goto statement found

This is not an error, just a suggestion to improve one's programming style. In most cases, Algol W programs can be made clearer and simpler by avoiding Goto statements and using other control statements (such as While and If statements) instead.

If the compiler parameter TERSE is in effect, the second form of the message appears for the first goto statement encountered; other occurrences are passed without warning.

Error 2052 - declarations must precede statements

In any block, all declarations must come before all executable statements. This includes procedure declarations as well as variable declarations. This message is sometimes caused by earlier syntax errors in declarations. Remedy: If the program is otherwise correct, just rearrange the declarations and statements in this block.

Error 2053 - substring length must be constant

The length of the substring (the quantity after the "|" or "//") must be an integer constant. Substrings must have constant lengths in Algol W, just like string variables.

Error 2054 - "<name>" is undeclared in this block

A variable with this name has been declared somewhere in the program, but the place where the variable is used (where the error

message occurs) is not within the scope of the declaration. Remedy: Move the declaration to some outer block.

Error 2055 - "array" missing in array declaration

The word 'array' must appear in an array declaration. Example:

```
integer A(1::10)
```

is incorrect.

```
integer array A(1::10)
```

is correct.

Error 2056 - error recovery fails; more errors may follow

The compiler has reported some errors in the program and is trying to continue, but it has detected inconsistencies in its internal tables. This is frequently caused by serious syntax errors in declarations. This is not a bug in the compiler; it is doing the best it can. Remedy: Fix the errors which the compiler found and recompile.

## PASS THREE ERROR MESSAGES

Pass 3 errors with numbers from 3001 to 3008 are disastrous, causing immediate termination of the compilation. After any fatal pass 3 error, a table of triples (coordinate number, byte offset, byte length), is listed, indicating how much code was generated for each statement in the current program segment. The last entry of this table and the last two byte lengths are occasionally not meaningful.

Pass 3 errors with numbers of 3009 or greater cause part of the compiled program to be inconsistent. Usually this is because a predeclared procedure argument error has been noticed which could not be detected in passes 1 or 2 because it did not violate the rules of Algol W grammar. A run time error 5023 will occur if this path through the program is executed. Compilation continues after the error message has been issued, and a complete object deck will be output.

Error 3001 - program segment overflow

This error message occurs because of a design constraint of the compiler: the total amount of machine code and constants for any procedure or other block with declarations must be less than 8192 bytes (a segment of code). All of the constants for a block are allocated in front of the first statement. Therefore, if the byte offset of the first statement is very large, constants are taking up too much space. This sometimes happens in programs with many

string constants (ten 80 character string constants take up 800 bytes). It is necessary to reduce the number of statements and/or constants in the block; this can be achieved by introducing new procedures or by inserting at least one declaration into some internal block(s), thereby forcing part of the block that was too big into more than one segment of code.

Error 3002 - compiler stack overflow

A push-down stack, used by the compiler while generating code, has overflowed. A program segment overflow was probably imminent. The remedies suggested in Error 3001 apply.

Error 3003 - compiler logic error

Internal consistency checks performed by the compiler have failed. Leave your card deck or source file exactly as it is. Take the program listing and card deck (or terminal output) to the appropriate member of the Computing Center staff.

Error 3004 - program area overflow

There is insufficient space in memory to contain the compiled program. Recompile with a larger SIZE parameter.

Error 3005 - data segment overflow

The data for each procedure or block with declarations are limited to 4096 bytes. Read the suggestions in error message 3001.

Error 3006 - coordinate table overflow

The table being constructed to supply the coordinate number in run-time error messages has overflowed. Recompile with a larger SIZE parameter.

Error 3007 - too many procedure calls

References to only 63 procedures are allowed within any single procedure. In this context, procedure means a code segment, and includes programs, procedures, non-trivial blocks, and external references.

Error 3008 - too many procedure parameters

A maximum of 34 arguments are allowed in the call of a procedure. This error message is also produced when the Call predeclared procedure is used to call a subroutine with more than 48 parameters.

Error 3009 - expression in get list

In a call to Get, Geton, or Getstring the third or subsequent argument specifies an expression. Such arguments designate the destination of a value obtained as the result of an input conversion, and must specify a variable. The statement:

```
Get(3, "2X,I8", Var*3);
```

would produce this error message.

Error 3010 - literal constant in get list

Read the comments for error 3009. A literal constant may not be specified as the destination for an input value. If allowed, such an action would overwrite program object code. The statement:

```
Get(4, "F8.4", 1.23);
```

would result in this error message.

Error 3011 - invalid predeclared procedure nesting

While the grammar of the language allows a predeclared procedure call as an argument to a predeclared procedure, in certain cases run time system restrictions make this impossible. For instance:

```
Put(6, "H0,I10", Get(5, null, A), A);
```

which is legal in the grammar cannot be handled by the run time system.

This error message results if a direct call of this kind is issued and can be detected by the compiler. If the call is indirect, such as a procedure called as an argument subsequently calling another predeclared procedure, then the compiler cannot detect this. The error is reported at run time instead. See the description of error 5030.

Error 3012 - procedure needs string parameter

In a call to one of the string input/output predeclared procedures a particular argument was expected of simple type string and was not found to be so. For example:

```
integer Card;
 .
Readcard(Card);
```

Predeclared procedures which can cause this error message are Readcard, Writecard, Getcard, Putcard, Xgetcard, and Xputcard.

Error 3013 - invalid format designator

A format designator is the second argument to one of the predeclared procedures Get, Geton, Getstring, Put, Puton, or Putstring. It must be either a string expression or the 'null' constant reference.

Error 3014 - procedure needs more parameters

Certain predeclared procedures need a minimum of two arguments; others need a minimum of three. Failure to provide at least this number will produce this error message. See the specification of the predeclared procedure in question in the main manual.

Error 3015 - predeclared procedures nested too deeply

This message should never occur. It is a consequence of a consistency check performed by the compiler during processing of predeclared procedures. A limit of 31 levels of nesting for predeclared procedure or predeclared function calls are allowed.

Error 3016 - parameters supplied with register call

An Rcall statement has more than one argument. Parameters supplied during an R-call subroutine call are taken from the predeclared variables R0 and R1.

Error 3017 - Sense key is not a string

A Sense statement has a second argument which is not a string expression. This second argument supplies keywords which request information about an input/output stream.

Error 3018 - invalid external symbol designator

An external symbol designator is the first argument of Call, or the single argument of External or Rcall, which specifies the external (loader) symbol definition name (ESDname) of a subroutine or data area. It must be given as a string constant from one to eight characters in length with no embedded blanks.

Error 3019 - procedure needs integer index

A predeclared procedure argument must be of simple type integer, and was not. For example, the second argument to Xgetcard should be an integer line number.

Error 3020 - Link must be nested in Call

An occurrence of the Link predeclared function was found which was not an argument of the Call predeclared procedure. The following sequence:

```
integer Ptr;
Ptr := Link("Myproc");
Call("SUBR", ..., ..., Ptr, ...);
```

is not allowed, since Myproc may not be in scope when the call is issued. Instead:

```
Call("SUBR", ..., ..., Link("Myproc"), ...);
```

is the required method of coding the call.

Error 3021 - invalid procedure designator

An argument to Link is invalid. The procedure designator must be a string constant containing the name of the procedure to be linked in, with no embedded blanks and between one and 32 characters in length.

Error 3022 - integer or bits parameter needed

A predeclared procedure argument must be of simple type integer or bits, and was not. For example, the second argument to Locate is an integer or bits variable which will receive a machine address.

Error 3023 - facility not allowed in monitor mode

This error should not normally occur and is documented here only for completeness.

## LOADER ERROR MESSAGES

Loader error messages are all of the form:

Error 4xxx - message

With the exception of 4009, which is a warning, all of these errors are disastrous and terminate processing.

Error 4001 - unrecognized loader record

An object card has been encountered which is not one of the set recognized by the loader. This should not occur; consult a member of the Computing Center staff.

Error 4002 - library search not available in monitor mode

This error should not normally occur and is documented here only for completeness.

Error 4003 - section definition not 1st esd item on card

An unexpected ordering of ESD items has been noticed by the loader. This should not occur; consult a member of the Computing Center staff.

Error 4004 - symbol on "RLD" card not defined

A symbol whose address is to be determined by the loader cannot be found. This should not occur; consult a member of the Computing Center staff.

Error 4005 - insufficient space to load program

Object programs generated by the compiler are loaded into the storage area vacated by the compiler tables. If this is not large enough, there will not be enough room. Recompile with a larger SIZE parameter.

Error 4006 - insufficient space to run program

Read the comments for error 4005. After the program has been loaded, remaining storage is allocated to the program for data and record storage during execution. If less than six pages remain, the loader will refuse to accept the program. Recompile with a larger SIZE parameter.

Error 4007 - section definition name repeated

Two program segments have been found with the same section definition name. This should not occur; consult a member of the Computing Center staff. Because the MTS loader overwrites, if possible, new definitions of routines over the old image, multiple definitions of external routines cannot be detected. A warning message is printed by MTS only if a subsequent definition is longer. Beware...

Error 4008 - symbols are undefined

One or more external (loader) symbols are undefined. A program references precompiled Algol W procedures or FORTRAN (or other O/S Type I) subroutines, and the object for these routines has not been supplied. It should have been given via a LIBSEARCH=<filename> compiler parameter.

Note: This message will normally only appear if CANCEL is given as a reply to the MTS undefined symbol prompt, or if *ALGOLW has been invoked in batch.

Error 4009 - warning: low core symbol table undefined

A low core symbol (LCS) table card has been encountered which specifies an unknown symbol. This should not occur; consult a member of the Computing Center staff.

Error 4010 - address constant length not 4 bytes

A short address constant has been encountered. This should not occur; consult a member of the Computing Center staff.

Error 4011 - card out of sequence

The Algol W loader expects object module cards to be submitted in the sequence ESD-TXT-RLD-END, and a loader card has been found out of sequence. This should not occur; consult a member of the Computing Center staff.

## RUN-TIME ERROR MESSAGES

All run-time error messages are in one of the two forms:

Run error 5xxx near <location> in <name> - message

or:

Exception 5xxx near <location> in <name> - message

The first form indicates a fatal run error; execution of the program will be terminated after the message and any post-mortem dump information has been printed.

The second form, distinguished by the word "exception", indicates an exceptional condition detected by Algol W and processed in accordance with the value of the relevant predeclared reference variable. This value will have been assigned to the variable in the user program. Execution of the program is restarted after the message has been printed.

Any exceptional condition which will terminate execution prints the first form of error message.

In the error message, <location> is a source program coordinate near to the cause of the error, and <name> is the name of the procedure which contains this coordinate. If the procedure is a main program, the characters "MAIN" are printed.

Under certain conditions Algol W cannot determine the location of a run time error or exception, so in these cases,

... near <location> in <name> ...

is replaced by:

... at unknown location ...

Programs compiled with the NOCHECK compiler parameter can never report a run time location.

## Fatal Run Error Conditions

Error 5001 - run parameter list

An error has been detected in the scanning of run time time parameters. These come from:

(1) the right hand side of a RUNPARM compiler parameter;

(2) parameters supplied at run time, from a /EXECUTE control record in compile, load and go mode, or via the MTS $RUN PAR= field with a program run from an object deck.

Error 5002 - case selection indexing

An index in a Case statement or Case expression is less than one or greater than the number of cases. Check for correct assignment of the variable or variables forming the expression.

Error 5003 - substring indexing

The substring selected extends off one end of the string. All of the characters specified by a substring designator must lie within the bounds of the string. Check the assignments to the variable or variables, if any, forming the offset expression. Does the addition of the constant length specify a character beyond the end of the string?

Error 5004 - assignment to name parameter

An attempt has been made to assign to an actual parameter which is not a variable, but is instead an expression, a constant, or a control identifier. When a procedure assigns a new value to a name parameter, the assignment takes place directly to the actual parameter supplied in the procedure call. That actual parameter must then be a variable, that is a quantity which would be legal on the left hand side of an assignment statement.

Error 5005 - data area overflow

No more storage is left for variables. This can happen if a procedure gets into a loop calling itself recursively, or if there really is not enough memory. In the latter case, running the program with a larger SIZE parameter may cure the trouble.

The message information includes the size of the run time work space, which is shared by both the data stack and record storage.

Before this message is produced Algol W will attempt to reclaim more space for the data area stack by compacting the record storage area. Only if this is unsuccessful will this message appear.

Error 5006 - actual-formal mismatch in procedure call

An actual parameter passed to a procedure is not assignment compatible with the corresponding formal parameter. The number of the parameter in error is printed with this message. Check this parameter against the declaration of the procedure named as the actual parameter for the formal procedure referenced by the erroneous call.

Error 5007 - record storage area overflow

No more storage exists for records. Running the program with a larger SIZE parameter may cure the trouble.

The message information includes the size of the run time work space, which is shared by both the data area stack and record storage.

Before this message is produced Algol W will attempt to reclaim more space for record storage by compacting the existing record storage area. Only if this is unsuccessful will this message appear.

Error 5008 - length of string input

The string read in is longer than the length of the receiving string variable or substring. Possibly a quote (") or prime (') has been omitted in the data, or two adjacent strings have no separating blank causing two quotes or two primes to be interpreted as a single character within the first string. Check your data.

Error 5009 - Link function error

An error condition has been detected during execution of the Link function, which provides the facility to call back main code procedures from a called FORTRAN subroutine. Certain restrictions are imposed, and the message gives details of the violation detected.

Error 5010 - input conversion failed

An input data item could not be converted to the type of the destination variable. The message gives details of the type of conversion requested, the data item in error, and the stream assignment from which it was input.

Error 5011 - reference input

Reference values may not be input. Input assignments to records must be made to the individual record field designators.

Error 5012 - reference output

Reference values may not be output. Output of records must be done by specifying the individual record field designators.

Error 5013 - incorrect number of parameters

The number of actual parameters in a procedure call is different from the number of formal parameters declared in the called procedure. Check the declaration of the procedure named as the actual parameter for the formal procedure referenced by the erroneous call.

Error 5014 - array too large

Either the first n-1 dimensions of an array declaration define too many elements, or an attempt has been made to declare an array greater than one megabyte in size. The product of the first n-1 dimension lengths (<upper-bound> - <lower-bound> + 1) multiplied by the size of a single element must be less than 32768 .

The element sizes are:

```
logical - 1
integer, real, bits, reference - 4
long real, complex - 8
long complex - 16
string - as declared
```

For example:

```
long real array A(1::4, 1::5000);
long real array B(1::5000, 1::4);
```

Array A is correct; array B will produce this error message. 5000 times the element length (eight for long real) is 40000, which is greater than the maximum of 32767 allowed.

Error 5015 - array subscripting

An array subscript is not within the declared bounds of the array. Check for correct assignment of the variable or variables forming the subscript expression or expressions.

Error 5016 - real to integer conversion failed

A real value whose modulus is greater than the maximum possible integer, 2147483647, cannot be converted to an integer value. Number representation is discussed in Appendix J.

Error 5017 - page estimate exceeded

The number of pages estimated has been exceeded. Page limit checking is on by default only for the PRINT stream (MTS SPRINT), but the program may have changed the default by calling the Qualify predeclared procedure.

The page limit in effect is that set by the PAGES compiler or run time parameters. In compile, load and go mode, if no PAGES parameter is given but a system limit is in force (MTS local or global page limit) then the Algol W system estimate is set to one less than the next system limit at the time the program commences execution. This ensures that Algol W diagnostic messages are printed before execution is stopped by the operating system.

Error 5018 - time estimate exceeded

The execution CPU time estimated has been exceeded.

The time limit in effect is that set by the ETIME or TIME compiler or run time parameters. In compile, load and go mode, if no parameter is given but a system limit is in force (MTS local or global time limit) then the Algol W system estimate is set to slightly less than the next system limit at the time the program commences execution. This ensures that Algol W diagnostic messages are printed before execution is stopped by the operating system.

Error 5019 - format string processing failed

One of the rules for the construction of format strings has been broken. The message printed gives the reason for rejecting the format string and the position within the string at which the error was detected. See the section "Format Directed Input and Output".

Error 5020 - format / type mismatch

An attempt has been made to input or output a value with a format code which is not compatible with the value's simple type. The message gives the simple type and format code involved. See the section "Format Directed Input and Output".

Error 5021 - incompatible field designator

An attempt has been made to access a field of a record using a reference which does not designate a record of the corresponding record class. The reference might be null or undefined. If it is not, check the record class used for its most recent assignment.

Error 5022 - assertion failed

The result of the evaluation of the logical expression in an Assert statement was false or undefined. The value of the assertion counter predeclared variable A_Count, which is incremented by one by each assert statement processed, is printed with the message.

Error 5023 - error marker encountered

An attempt has been made to execute a path through the program which was flagged as invalid by the code generation phase of the compiler.

This error will be produced at run time if one of the non-fatal pass three compiler errors has been given for the section of program in question. Marking such sections of code as invalid allows the program to be run safely to test other paths through the program without recompilation.

Error 5024 - no read allowed on this I/O stream

An input/output stream could not be accessed from an input statement.

(1) One of the named predefined output streams PRINT, PUNCH, or ERROR has been used in an input statement.

(2) The file-or-device attached to the specified input/output stream may not be accessed to read data in. If a file, check that it is permitted for read access.

Error 5025 - no write allowed on this I/O stream

An input/output stream could not be accessed from an output statement.

(1) One of the named predefined input streams INPUT or USER has been used in an output statement.

(2) The file-or-device attached to the specified input/output stream may not be accessed to write data. Did you intend to write to it? If a file, check that it is permitted for write access. If a magnetic tape, check that it was mounted with the necessary write-permit ring.

Error 5026 - rewind of this I/O stream is not allowed

An attempt has been made to rewind a file or device which is not rewindable.

(1) One of the named predefined streams INPUT, PRINT, PUNCH, ERROR, or USER has been used in a Rewind statement. These streams may not be rewound.

(2) The file-or-device attached to the specified input/output stream does not have rewind capability. Did you intend to rewind it? Devices such as terminals, card readers and printers cannot be rewound; files and tapes may be.

Error 5027 - invalid or undefined I/O stream

The stream designator specified in an input/output procedure call is either incorrectly constructed or is not defined. Algol W input/output stream names must be either predefined or user defined. If the latter, they must be defined by a call to Assign before being used in any other input/output predeclared procedure call. Valid stream names are constructed as follows:

(1) A string expression containing the name of a predefined input/output stream. Either:

(a) one of the named streams INPUT, PRINT, PUNCH, ERROR, or USER, or

(b) one of the numbered streams, 0 to 19

(2) An integer expression containing a value between 0 and 19, specifying a predefined numbered stream.

(3) One of the predeclared bits variables Input, Print, Punch, Error, or User, representing the corresponding named predefined stream.

(4) Either of the predeclared bits variables Rdr or Wtr, representing the basic reader or writer stream.

(5) A string expression containing the name of a user defined stream. This string must contain only alphanumeric characters, with no embedded blanks.

Error 5028 - Empty failed

An attempt has been made to empty a stream which is either not assigned, not a file, or for which the required access is not available.

(1) One the named predefined streams INPUT, PRINT, PUNCH, ERROR, or USER has been used in an Empty statement. These streams may not be emptied.

(2) The attached file-or-device is not a file; only disk files may be emptied.

(3) The file attached to the specified input/output stream may not be emptied. Did you intend to empty it? Check that you have the necessary write access to the file.

Error 5029 - too many I/O streams

A maximum of 25 user defined input/output streams may be simultaneously active. An attempt to create more than that number will produce this error message. User defined streams should be released when no longer needed by the program; have you forgotten to do so?

Error 5030 - predeclared procedures illegally nested

An attempt has been made to call an input/output predeclared procedure while a similar procedure call was being processed.

This error message indicates a condition trapped at run time because it could not be detected at compile time. See the comments for error 3011.

Has a proper or function procedure call been issued from within the argument list of an I/O predeclared procedure call? Does the called procedure contain an I/O procedure call?

Error 5031 - I/O error on read

A serious error (not end-of-file) has been detected while trying to input a record from the specified file-or-device. The system I/O subroutine return code is reported as part of the message.

Error 5032 - I/O error on write

A serious error has been detected while trying to output a record to the specified file-or-device. The system I/O subroutine return code is reported as part of the message.

Error 5033 - Assign failed

Either a file or device name was not given, or it could not be attached.

(1) The stream specified was one the named predefined streams INPUT, PRINT, PUNCH, ERROR, or USER. These particular streams may not be reassigned.

(2) All of the characters in a file-or-device name are blanks.

(3) A character string has been supplied which is not a valid MTS file-or-device name.

Error 5034 - Release failed

The file or device attached to the specified stream could not be released.

(1) The stream specified was one the named predefined streams INPUT, PRINT, PUNCH, ERROR, or USER. These particular streams may not be released.

(2) For any other valid input/output stream this error condition should not occur. Please consult a member of the Computing Center staff.

Error 5035 - indexed operation on this I/O stream not allowed

An attempt has been made to perform an indexed input/output operation using a stream which is not assigned, not a file, or for which the required access is not available.

(1) One of the named predefined streams INPUT, PRINT, PUNCH, ERROR, or USER has been used in a call to Xgetcard, Xputcard, or Xdelete. These streams may not be referenced by an indexed input/output operation.

(2) The attached file-or-device is not a file; indexed input/output is available only for disk files.

(3) The file attached to the specified input/output stream is of a type (such as sequential) which does not support indexed operations. Did you intend such an operation?

Error 5036 - Control failed

A control command passed to the operating system via the Control predeclared procedure has been rejected. The system return code for the failure is printed with the message.

Error 5037 - Qualify failed

A keyword or keyword expression given as a string argument to the Qualify predeclared procedure could not be recognized.

Error 5038 - Iocontrol failed

A keyword or keyword expression given as a string argument to the Iocontrol predeclared procedure could not be recognized.

Error 5039 - no Sense keyword

In the second argument supplied in a call to the Sense predeclared procedure, either no keyword was given, or the list of keywords has been exhausted.

Error 5040 - Sense keyword processing failed

A keyword or keyword expression given as a string argument to the Sense predeclared procedure could not be recognized.

Error 5041 - Sense keyword and data type inconsistent

In a call to the Sense predeclared procedure, a supplied keyword and the simple type of the corresponding receiving variable are incompatible.

Error 5042 - data driven replication factor for input

The special data driven replication factor "R" may not be specified for an input operation. This factor is intended for use in calls such as:

```
Put(Print, "4X,RX,RH*", Num_Spaces, Num_Asterisks);
```

where the length of a space or character field, or an output tab position, are supplied in put-list variables. The equivalent input operation is not supported.

Error 5043 - input field all break characters

In a format directed input operation, an attempt has been made to decode an input data item from a field composed entirely of break characters (blanks or commas).

Error 5044 - data item not alone in field

In a format directed input operation, an attempt has been made to decode an input data item from a field which contained more than one data item. Break characters (spaces or commas) are not allowed within a data item. For example:

```
Get(Input, "3I5", Ia, Ib, Ic);
```

with an input data line:

```
123 456 789
```

will fail while decoding the value of Ia since the I5 field will contain "123 4", instead of the expected single integer value.

Error 5045 - break characters in hexadecimal field

This error message is produced as a result of a condition detected during "Z" format input of a floating point value. Such values must be completely specified. For instance, a long real variable occupies 8 bytes of storage and requires exactly 16 hexadecimal digits ("0"-"9" or "A"-"F" or "a"-"f") to specify it.

Error 5046 - Hexadecimal input conversion failed

During a format directed input operation, "Z" format hexadecimal input conversion has failed. This means that a character has been found which is not a valid hexadecimal digit ("0"-"9" or "A"-"F" or "a"-"f").

Error 5047 - Bad parameter to NEWLINE

The predeclared procedure NEWLINE has been called with a parameter value that is less than zero or greater than 60.

Error 5048 - An Algol W string function procedure has been called from a non-Algol W routine.

Algol W string function procedures may not be called from procedures written in languages other than Algol W because there is no standard way of returning string values using the O/S Type I calling convention.

## Predeclared Function Exceptional Conditions

Run time error messages in the following ranges are the result of conditions detected during the evaluation of predeclared functions of analysis:

| | |
|---|---|
| 5201-5299 | real predeclared functions |
| 5301-5399 | long real predeclared functions |
| 5401-5499 | complex predeclared functions |
| 5501-5599 | long complex predeclared functions |

This is not to say that there are four hundred different error conditions; there are not. The information given with the error message is sufficient to identify the function in error and includes the invalid argument supplied to it. For this reason a complete list of error messages is not given here. The information in Appendix F includes the valid argument ranges for each function. In many cases this is considerably less the range of rational numbers which may be represented on the computer.

Predeclared functions of analysis may report error conditions for one of several reasons.

(1) Argument values for which the function is not defined; for instance Log(-1). The logarithm of a negative argument yields a complex result.

(2) Argument values which are singularities in the definition of a function; for instance Cot(0). The cotangent function is discontinuous at this argument value.

(3) Argument values for which the evaluation of the function yields a result which may not be represented on the computer; for instance Exp(500). The exponential function value for this argument is greater than machine 'infinity'. (Number representation is discussed in Appendix J.)

(4) Although the function is defined for the specified argument and the result representable on the computer, round-off would destroy the accuracy; for instance Sin(3'+17). The sine function is computed by reducing the argument to the equivalent in the range 0 to 2*pi. If this cannot be done with sufficient accuracy (because significance is lost) then the sine cannot be computed.

Predeclared function error messages normally start with the words "Run error" and cause a fatal termination of the executing program. This action can be overridden by assignments to the predeclared system reference variable Function. Where such an assignment causes the error condition to be recovered sufficiently so that execution will resume, the word "Exception" replaces "Run error" at the start of the message if one is printed.

## Program Interruption Exceptional Conditions

A program interruption occurs on a computer when a program attempts execution of an instruction which is either invalid, at an invalid location, or specifies one or more instruction operands which are invalid or at invalid locations.

For an example of this consider the following small section of an Algol W program:

```
real Dividend, Divisor, Quotient;
Read(Dividend, Divisor);
Quotient := Dividend / Divisor;
```

When the compiler generates object code for the last statement, a divide instruction is assembled as part of the object program output. When the program segment executes, however, the Read statement may supply a value of zero for the divisor. Division by zero has no defined result, so the computer halts the program at this point with a program interruption. Algol W can take control when such conditions occur, and an error

message in the 59xx series is printed. In this case the "xx" is the system's program interruption code; the explanation of the code is printed as part of the message.

The errors in this series are:

| Number | Interrupt |
|---|---|
| 5901 | operation |
| 5902 | privileged operation |
| 5903 | execute |
| 5904 | protection |
| 5905 | addressing |
| 5906 | specification |
| 5907 | data |
| 5908 | integer overflow |
| 5909 | integer division by zero |
| 5910 | decimal overflow |
| 5911 | decimal division by zero |
| 5912 | exponent overflow |
| 5913 | exponent underflow |
| 5914 | significance |
| 5915 | floating point division by zero |
| 5916 | software |

The message printed for these error conditions may also include certain location information. If the error message states that the condition occurred within an Algol W library routine, please contact a member of the Computing Center staff. (The 5916 message is a 'catch all' for any other condition or library error and should never occur.)

The message may state instead that the error occurred while executing an external routine. If so, the error occurred during the execution of a non-Algol W routine to which control has been passed via a Call or Rcall predeclared procedure or a FORTRAN statement. Algol W attempts to determine whether the called routine was coded in FORTRAN. If it was, the name of the called subroutine or function subprogram is printed with the error message. If not, the entry point address of the called routine is printed.

Error messages which do not specify either a library routine or an external location refer to an error condition during execution of Algol W object code. Certain of these error conditions may be intercepted by prior assignment to one the predeclared reference variables listed below:

| Variable | Condition |
|---|---|
| Intovfl | 5908 -- integer overflow |
| Intdivzero | 5909 -- integer division by zero |
| Ovfl | 5912 -- exponent overflow |
| Unfl | 5913 -- exponent underflow |
| Divzero | 5915 -- floating point division by zero |

When an error message is printed as a result of one of these conditions, and a prior assignment has been made to the relevant predeclared variable, the phrase "Run error" in the message is replaced by the word "Exception". Note also that occurrences of the exponent underflow condition will never be reported unless an assignment is made to the Unfl variable, since its value is initialized to 'null'.

An addressing exception within Algol W object code may be the result of a null or undefined reference; other conditions not in the predeclared variable list above should be reported to Computing Center staff.

## End-of-File Exceptional Conditions

This section refers to message number 5999.

End-of-file detected by the following predeclared procedures is treated as an exceptional condition:

Read
Readon
Readcard
Get
Geton

Note however that the following predeclared procedures do not treat end-of-file or record absent as an exceptional condition:

Getcard
Xgetcard

These two procedures only set the predeclared variable Filemark to 'true', and it is the responsibility of the program to test for end-of-file or record absent on return.

For the first list of predeclared procedures, end-of-file exceptions are processed under the control of the Endfile predeclared reference. By default end-of-file conditions cause a fatal run error message (number 5999) to terminate the program. If a prior assignment to the Endfile reference will cause the program to resume execution, then the phrase 'Run error' is changed to 'Exception' as with the other exceptional conditions.

# APPENDIX D: BASIC SYMBOLS

## CHARACTER SET

Algol W programs are written using the following character set.

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

a b c d e f g h i j k l m n o p q r s t u v w x y z

0 1 2 3 4 5 6 7 8 9

+ - * / = : ; . , # " ' | ¬ ( ) < > % _

Note that in reserved words and identifiers, described later, lower case letters are treated as if they were the equivalent upper case letter.

Within quotes ("), any character from the set of 256 valid EBCDIC codes is valid when defining a string constant.

## SPECIAL CHARACTER SYMBOLS

| Symbol | Description | Function |
|---|---|---|
| " | quotation mark | string quotes |
| # | hash mark | hexadecimal prefix |
| ' | prime | scientific notation for decimal exponentiation |
| \| | vertical bar | signifies substring |
| // | double slash | alternative to the vertical bar in a substring |
| , | comma | separates elements in a sequence |
| ; | semicolon | separates declarations and statements |
| : | colon | signifies preceding label |

| | | |
|---|---|---|
| . | period | decimal point or end of program |
| ( | left parenthesis | multiple uses |
| ) | right parenthesis | multiple uses |
| + | plus sign | unary and binary addition |
| - | minus sign | unary and binary subtraction |
| * | asterisk | multiplication or formal array parameter dimension |
| / | slash | division |
| ** | double asterisk | exponentiation |
| ¬ | not sign | logical negation |
| _ | underscore | possible character in identifier |
| = | equal sign | equality |
| ¬= | not sign followed by equal sign | not equal |
| < | less than sign | less than |
| <= | less than sign followed by equal sign | less than or equal to |
| > | greater than sign | greater than |
| >= | greater than sign followed by equal sign | greater than or equal to |
| :: | double colon | separates lower from upper array bound |
| := | colon followed by equal sign | assignment operator |
| % | percent sign | delimits a brief comment; such comments may also be terminated by a semicolon |

## RESERVED WORDS

The following character sequences are reserved words of the Algol W language; they may not be used as identifiers.

| Reserved Word | Function |
|---|---|
| abs | operator; absolute value of operand |
| algol | defines precompiled procedure 'body' |
| and | operator; logical and of operands |
| array | used in subscripted variable declarations |
| assert | used in Assert statement; causes fatal exit if following logical expression is 'false' |
| begin | starts a block, or group, of statements |
| bits | simple type designator in declarations |
| case | case selection; Case statement or expression |
| comment | starts a comment; ended by next semicolon |
| complex | simple type designator in declarations |
| div | arithmetic operator; integer division of operands without remainder |
| do | ends a 'while' or 'for' clause |
| else | starts an 'else' clause in an If statement or expression |
| end | ends a block of statements started by a previous 'begin' |
| false | logical constant |
| for | iteration; declares a local control integer variable which is incremented until a limit is reached |
| fortran | obsolete; see Call predeclared procedure defines a subroutine as a procedure body |
| goto<br>go to | Goto statement; unconditional jump<br>'to' is a reserved word only if it follows 'go' |

| | |
|---|---|
| if | conditional execution; If statement or expression |
| integer | simple type designator in declarations |
| is | operator; returns logical 'true' if both reference operands are bound to the same record class |
| logical | simple type designator in declarations |
| long | modifies 'real' or 'complex' declarations to be long precision simple type designators; as an operator the operand is converted to long precision |
| not | operator; logical negation; this reserved word is exactly equivalent to the special character, ¬ |
| null | reference constant; points to no record |
| of | ends 'case' selection clause |
| or | operator; the logical or of operands |
| procedure | type designator in declarations; declares either the start of a logically grouped set of statements, or a formal procedure parameter within a proper or function procedure heading |
| real | simple type designator in declarations |
| record | type designator in declarations; declares a data structure |
| reference | simple type designator in declarations; declares a pointer to a data structure defined in a record class declaration |
| rem | arithmetic operator; the remainder after the integer division of the operands |
| result | qualifies a simple type parameter in a procedure heading; indicates an expected returned value |
| shl | operator; defines a logical left shift operation |
| short | arithmetic operator; the operand is converted to short precision |
| shr | operator; defines a logical right shift operation |
| step | precedes an optional increment in a For statement |

| | |
|---|---|
| string | simple type designator in declarations |
| then | ends an 'if' clause conditional expression |
| to | see 'goto'; (not strictly a reserved word) |
| true | logical constant |
| until | precedes a For statement limit |
| value | qualifies a simple type parameter in a procedure heading; indicates a value passed within the parameter (as opposed to a name parameter) |
| while | iteration statement; terminates when subject condition becomes 'false' |

## IDENTIFIERS

An identifier is comprised of between 1 and 256 characters. The first character of an identifier must be alphabetic (A-Z, a-z). Subsequent characters may be any mixture of alphabetic, numeric (0-9), or the underscore character (_).

When lower case alphabetics are encountered in an identifier they are assumed to represent the equivalent upper case character. Differences in case only do not distinguish two otherwise identical identifiers.

Reserved words may not be used as identifiers.

In the Algol W environment, a number of identifiers are predefined (Pi, Write, Log, etc.). These identifiers may be assumed to be declared in a conceptual block enclosing the entire Algol W program. They may therefore be redeclared in the user program if so desired. However, if this is done the system meaning of the symbol will not be available in the scope of the redeclared identifier.

## EMPTY SYMBOL

The symbol <empty>, where <empty> is no physical symbol, is also considered an element of the Algol W set of symbols.

# APPENDIX E: PREDECLARED PROCEDURES

This appendix gives a summary of all predeclared procedures defined within the Algol W environment. The list is in alphabetical order of procedure name.

All of these predeclared procedures may be assumed to be declared as Algol W proper procedures in an outer block enclosing the user source program. However any type checking of parameters is done as a result of the implementation; it is not forced by the grammar of the language. The identifier names assigned to these procedures may be redeclared within the user program block structure.

All procedures listed here are fully described in the main manual.

In the summaries which follow:

(1) angle brackets, < >, specify an item which should be replaced by an actual parameter;

(2) square brackets, [ ], specify an optional item;

(3) braces, { }, specify a set of alternative items. These alternatives are separated by vertical bars, |.

Assign(<stream>, <fdname>)

The Algol W input/output stream designated by the first parameter is attached to the file-or-device specified by the second in the form of a string expression. Any previous file-or-device so attached is released first. The named predefined streams INPUT (SCARDS), PRINT (SPRINT), PUNCH (SPUNCH), ERROR (SERCOM), and USER (GUSER) may not be reassigned.

Attntrap(<logical-expression>)

If the logical expression evaluates to 'true', attention interrupt trapping is enabled. If it is 'false', the trap is disabled. While the trap is enabled, attentions do not interrupt program execution; instead the predeclared logical variable Attnmark is set to 'true'. It is the responsibility of the user program to inspect and act on the value of this variable.

Call(<esdname> [, <parameters>])

The external O/S Type I linkage subroutine designated by the first parameter is called. This parameter must be:

(1) a string constant from 1 to 8 characters in length, specifying the entry point external symbol definition name of the desired subroutine; or

(2) an integer or bits expression specifying the entry point address.

The parameters given may be of any simple type, but should correspond to those required by the called subroutine. For a full description of this procedure see the section "External Linkages"; the O/S Type I linkage is described in Appendix K.

Cmd(<string-expression>)

The string expression is evaluated and passed to the system for execution as an MTS command. Control will return to the next statement in the program provided that the command executed does not cause unloading of the program.

Control(<stream>, <string-expression>)

The string expression is evaluated and sent to the system as a control command for the input/output stream specified by the first parameter. This allows system control commands for terminals, tapes and files to be issued under program control.

Empty(<stream>)

The file, if any, attached to the designated input/output stream is emptied if possible. The named predefined streams INPUT (SCARDS), PRINT (SPRINT), PUNCH (SPUNCH), ERROR (SERCOM), and USER (GUSER) may not be emptied.

Fetch(<source-address>, <target-variable> [, <length>])

Use with care. The first parameter should be an integer or bits expression specifying a machine address. The second parameter may be any Algol W variable. Bytes are copied from the address designated by the first parameter to the variable designated by the second. The number of bytes moved defaults to the implied length of the target variable, but may be explicitly given by an integer expression supplied as the optional third parameter.

Flush(<stream>)

Any pending information in the output buffer for the designated input/output stream is immediately written to the attached file-or-device.

Get(<stream>, {<format-string> | null} [, <get-list>])

Provides format-directed or free-format input operations from the input/output stream designated by the first parameter. The second

parameter specifies the format to be used (if a string expression) or that the operation is to proceed in free format (if it is the null reference). The get-list is a list of one or more variables, separated by commas, which are assigned values from left to right as a result of the input operation. This get-list is optional because a format string may merely specify a positional operation (tabbing or skipping of records). Each Get will fetch a new physical input record.

Getcard(<stream>, <string-variable>)

A single physical input record is fetched from the stream designated by the first parameter, and the bytes are placed in the string variable without editing (truncated or padded with blanks on the right as necessary). Note: for this procedure it is the responsibility of the user program to test for end-of-file by inspecting the predeclared logical variable Filemark on return.

Geton(<stream>, {<format-string> | null} [, <get-list>])

Geton is similar to Get; the only difference is that while Get will always fetch a new physical input record, Geton will only do so if the information on the current record is exhausted (or a format string specifies such action).

Getstring(<string-expression>, {<format-string> | null} [, <get-list>])

The conversion operations performed by Getstring are similar to those of Get; the difference lies in the first parameter which for this procedure is a string expression which takes the place of a physical input record. Error detection by the user program is possible with this procedure - see main text description.

Iocontrol({<integer-expression> | <string-expression>})

Action depends on the value of the parameter; for each integer recognized there is an equivalent keyword character string. If a string expression is specified it is assumed to consist of one or more keywords separated by blanks or commas. Iocontrol allows:

(1) control of basic input/output operations;

(2) selection of termination timing information display;

(3) selection of program interrupt information display;

(4) modification of Getstring action;

(5) modification of the string recognition algorithm.

Locate(<variable>, <integer-or-bits-variable>)

Use with care. The second parameter variable is set to the machine address of the first parameter variable. Avoid use of the address of a variable when that variable is not in scope.

Move(<source-variable>, <target-variable> [, <length>])

Use with care. The first and second parameters may specify any Algol W variables. Bytes are copied from the variable designated by the first parameter to the variable designated by the second. The number of bytes moved defaults to the minimum of the two implied lengths of the variables, but may be explicitly given by an integer expression supplied as the optional third parameter. No type conversion is done by this procedure.

Newline({<integer-expression> | <string-expression>})

Skips one or more lines on the basic output stream.

(1) If an integer expression is given, that many lines are skipped, subject to a maximum of sixty. If the expression value is zero, an overstruck line is begun.

(2) If a string expression is supplied, a new line is begun with the carriage control character set to the value of the first character of the string.

Protect(<stream>)

If any information is currently held in the output buffer for the designated stream, it is flushed out to the attached file-or-device. If the stream is attached to a file then the system virtual file buffers are written to disk; that is the disk copy of the file is brought up to date. This might be required at critical stages in file processing to protect against the effects of a system failure.

Put(<stream>, {<format-string> | null} [, <put-list>])

Provides format-directed or free-format output operations to the input/output stream designated by the first parameter. The second parameter specifies the format to be used (if a string expression) or that the operation is to proceed in free format (if it is the null reference). The put-list is a list of one or more expressions, separated by commas, whose values are output from left to right as a result of the output operation. This put-list is optional because a format string may merely specify a positional operation (tabbing or skipping of records). Each Put will start a new physical output record.

Putcard(<stream>, <string-expression>)

A single physical output record is sent to the stream designated by the first parameter, comprising the bytes from the string expression. Any information in the output buffer will be flushed to the assigned file-or-device name first.

Puton(<stream>, {<format-string> | null} [, <put-list>])

Puton is similar to Put; the only difference is that while Put will always start a new physical output record, Puton will only do so if the output buffer becomes full (or a format string specifies such action).

Putstring(<string-variable>, {<format-string> | null} [, <put-list>])

The conversion operations performed by Putstring are similar to those of Put; the difference lies in the first parameter which for this procedure is a string variable which takes the place of a physical output record.

Qualify(<stream>, <string-expression>)

Sets attributes of the input/output stream designated by the first parameter according to a keyword or keywords contained in the string expression. An example of its use would be to change the input or output lengths used by Algol W to determine input/output buffer overruns.

Rcall(<esdname>)

The external O/S Type I linkage subroutine designated by the single parameter is called. This procedure is similar to Call but provides for the case where parameters are expected in machine general registers. The ESDname is as specified for Call. General registers zero and one are loaded from the predeclared integer variables R0 and R1 before control is transferred to the subroutine.

Read(<read-list>)

Provides free-format input operations from the basic input stream. The read-list is a list of one or more variables, separated by commas, which are assigned values from left to right as a result of the input operation. Each Read will fetch a new physical input record.

Readcard(<string-variable>)

A single physical input record is fetched from the basic input stream, and the bytes are placed in the string variable without editing (truncated or padded with blanks on the right as necessary).

Reader(<stream>)

The input/output stream designated becomes the basic input stream. Initially the basic input stream is INPUT (MTS SCARDS).

Readon(<read-list>)

Readon is similar to Read; the only difference is that while Read will always fetch a new physical input record, Readon will only do so if the information on the current record is exhausted.

Release(<stream>)

Any file-or-device attached to the designated stream is released. If the stream is a predefined one, it is unassigned on return. If the stream is a user defined one, it is deleted from the system tables. This procedure is intended for use with Assign to dynamically access files-or-devices, in that it frees them when no longer needed. The named predefined streams INPUT (SCARDS), PRINT (SPRINT), PUNCH (SPUNCH), ERROR (SERCOM), and USER (GUSER) may not be released.

Rewind(<stream>)

The file-or-device attached to the designated stream is rewound. Note that in MTS only the currently active member of a concatenation will be rewound. The named predefined streams INPUT (SCARDS), PRINT (SPRINT), PUNCH (SPUNCH), ERROR (SERCOM), and USER (GUSER) may not be rewound.

Sense(<stream>, <string-expression>, <sense-list>)

Information about the stream designated by the first parameter is returned according to requests by keywords supplied in the string expression. The sense-list is a list of variables which receive the information by assignment from left to right as it is returned. Sense provides an input operation, as does Get, but the data returned is information about the attached file-or-device rather than the result of a physical read operation from it. Sense can be used to check the validity of an operation (such as indexed input/output or rewinding) before issuing the relevant request.

Stop({<string-expression> | null})

Execution of the program is immediately terminated. If the parameter is a string expression, the result of its evaluation is printed on ERROR (MTS SERCOM).

Store(<source-variable>, <target-address> [, <length>])

Use with care. The first parameter may be any Algol W variable. The second parameter should be an integer or bits expression specifying a machine address. Bytes are copied from the variable

designated by the first parameter to the address designated by the second. The number of bytes moved defaults to the implied length of the source variable, but may be explicitly given by an integer expression supplied as the optional third parameter.

Trace(<expression>)

This procedure currently does nothing; it may not remain so.

Translate(<source-variable>, <translate-table> [, <length>])

Use with care. The first parameter may designate any Algol W variable, but should normally be a string. The second parameter should either specify a string(256) expression, or an integer or bits expression whose value is the machine address of a 256-byte region. This second parameter is a translate table, and bytes within the source variable are translated in situ by reference to it. The number of bytes translated defaults to the implied length of the source variable, but may be explicitly given by an integer expression supplied as the optional third parameter.

Write(<write-list>)

Provides output operations to the basic output stream. The write-list is a list of one or more expressions, separated by commas, whose values are output from left to right as a result of the output operation. Each Write will start a new physical output record. The output operations are under the control of a set of predeclared format variables.

Writecard(<string-expression>)

A single physical output record is sent to the basic output stream, comprising the bytes from the string expression. Any information in the output buffer will be flushed to the assigned file-or-device name first.

Writeon(<write-list>)

Writeon is similar to Write; the only difference is that while Write will always start a new physical output record, Writeon will only do so if the output buffer becomes full (or the format specifies such action).

Writer(<stream>)

The input/output stream designated becomes the basic output stream. Initially the basic output stream is PRINT (MTS SPRINT).

Xdelete(<stream>, <integer-expression>)

The physical data record located at the line number given by the second parameter is deleted from the file attached to the stream designated by the first parameter.

Xgetcard(<stream>, <integer-expression>, <string-variable>)

Fetches a complete physical input record, like Getcard, but the operation is an indexed one. The record is fetched from a line number specified by the second parameter in the file attached to the stream designated by the first parameter. If no record exists at the specified location, the condition is treated as end-of-file. As with Getcard, the user program must test for this condition by inspecting the value of the predeclared logical variable Filemark.

Xputcard(<stream>, <integer-expression>, <string-expression>)

Sends a complete physical output record, like Putcard, but the operation is an indexed one. The record is sent to a line number specified by the second parameter in the file attached to the stream designated by the first parameter.

# APPENDIX F: PREDECLARED FUNCTIONS

This appendix gives a summary of all predeclared functions defined within the Algol W environment. The list is in alphabetical order of function name, with the exception that the long precision versions of floating point functions will be found listed with the short precision name. For instance, Longsin is listed with Sin.

All of these predeclared functions may be assumed to be declared as Algol W function procedures in an outer block enclosing the user source program. They may be redeclared within the user program block structure.

All functions listed here are fully described in the main manual, but domain of definition information (where applicable) is provided in this list only.

```
real procedure Arccos(real value X);
long real procedure Longarccos(long real value X);
```

Returns the inverse cosine of X.
Domain of definition of these functions:

```
     Arccos :        |X| <= 1.0
     Longarccos :    |X| <= 1.0
```

```
real procedure Arcsin(real value X);
long real procedure Longarcsin(long real value X);
```

Returns the inverse sine of X.
Domain of definition of these functions:

```
     Arcsin :        |X| <= 1.0
     Longarcsin :    |X| <= 1.0
```

```
real procedure Arctan(real value X);
long real procedure Longarctan(long real value X);
```

Returns the inverse tangent of X.
Domain of definition of these functions:

```
     Arctan :        all values of X
     Longarctan :    all values of X
```

```
string(12) procedure Base10(real value X);
string(20) procedure Longbase10(long real value X);
```

Both of these functions return a string encoding of X in radix 10 (decimal) form. The formats returned are:

```
Base10 :        "b+EE-DDDDDDD"
Longbase10 :    "b+EE-DDDDDDDDDDDDDDD"
```

where "b" is a blank; "+" is a sign (blank for a zero or positive mantissa); "E" is an exponent decimal digit; and "D" is a mantissa decimal digit.

```
string(12) procedure Base16(real value X);
string(20) procedure Longbase16(long real value X);
```

Both of these functions return a string encoding of X in radix 16 (hexadecimal) form. The formats returned are:

```
Base16 :        "bb+BB+AAAAAAA"
Longbase16 :    "bb+BB+AAAAAAAAAAAAAAA"
```

where "b" is a blank; "+" is a sign (blank for a zero or positive mantissa); "B" is an exponent hexadecimal digit; and "A" is a mantissa hexadecimal digit.

```
bits procedure Bitstring(integer value N);
```

Returns a bits value with the same internal representation as the integer N parameter. The returned value is the two's complement representation of N. Number representation is discussed in Appendix J.

```
string(1) procedure Code(integer value N);
```

Returns a string(1) value with the numeric code given by abs(N rem 256); see Appendix B for a list of integer codes corresponding to printable characters.

```
real procedure Cos(real value X);
long real procedure Longcos(long real value X);
```

Returns the cosine of X.
Domain of definition of these functions:

```
Cos :       |X| < (2**18)*pi = 823549.625
Longcos :   |X| < (2**50)*pi = 3.537118'+15
```

```
real procedure Cosh(real value X);
long real procedure Longcosh(long real value X);
```

Returns the hyperbolic cosine of X.
Domain of definition of these functions:

```
Cosh :        |X| < 175.3662
Longcosh :    |X| < 175.3662
```

```
real procedure Cot(real value X);
long real procedure Longcot(long real value X);
```

Returns the cotangent of X.
Domain of definition of these functions:

```
Cot :         |X| < (2**18)*pi = 823549.625
Longcot :     |X| <= (2**50)*pi = 3.537118'+15
```

Both functions exclude the singularities at:

```
X = n*pi,
```

where n = ..., -1, 0, 1, ...

```
complex procedure Cxcos(complex value Z);
long complex procedure Longcxcos(long complex value Z);
```

Returns the complex cosine of Z.
Let Z = X+Yi; domain of definition of these functions:

```
Cxcos :        |X| < 823549.625;  |Y| < 174.673
Longcxcos :    |X| < 3.537118'+15;  |Y| < 174.673
```

```
complex procedure Cxexp(complex value Z);
long complex procedure Longcxexp(long complex value Z);
```

Returns the complex exponential of Z.
Let Z = X+Yi; domain of definition of these functions:

```
Cxexp :        -180.2182 < X < 174.673;  |Y| < 823549.625
Longcxexp :    -180.2182 < X < 174.673;  |Y| < 3.537118'+15
```

If undeflow is not being trapped (the default case) then the X value may be less than -180.2182 and the real part of the result is returned as zero. See the comments for Exp and Longexp.

```
complex procedure Cxln(complex value Z);
long complex procedure Longcxln(long complex value Z);
```

Returns the complex natural logarithm (base e) of Z.
Domain of definition of these functions:

```
Cxln :        excludes the singularity at Z = 0.0
Longcxln :    excludes the singularity at Z = 0.0
```

```
complex procedure Cxsin(complex value Z);
long complex procedure Longcxsin(long complex value Z);
```

Returns the complex sine of Z.
Let Z = X+Yi; domain of definition of these functions:

```
Cxsin :        |X| < 823549.625;  |Y| < 174.673
Longcxsin :    |X| < 3.537118'+15;  |Y| < 174.673
```

```
complex procedure Cxsqrt(complex value Z);
long complex procedure Longcxsqrt(long complex value Z);
```

Returns the complex square root of Z.
Let Z = X+Yi; domain of definition of these functions:

```
Cxsqrt :        all values of Z
Longcxsqrt :    all values of Z
```

```
string(24) procedure Date(integer value N);
```

Returns a string encoding of the time and date, in a format determined by N. This is best illustrated by examples of strings returned:

```
Date(0) : " 15:08:40 03-25-80 3 085"
Date(1) : "  3:08 pm  Tue 25 Mar 80"
```

Both of these examples show the same time. Note that in the N = 0 'data' format, the date is given in the United States form month-day-year; 3 is the day of the week (Tuesday); and this is the 85th day of the year.

The result for any other value of N is undefined.

```
integer procedure Decode(string(1) value S);
```

Returns the numeric code for the character S; see Appendix B for a list of integer codes corresponding to printable characters.

```
integer procedure Entier(real value X);
```

Returns an integer value I such that: I <= X < I + 1

```
real procedure Erf(real value X);
long real procedure Longerf(long real value X);
```

Returns the error function of X.
Domain of definition of these functions:

```
Erf :        all values of X
Longerf :    all values of X
```

```
real procedure Erfc(long real value X);
long real procedure Longerfc(long real value X);
```

Returns the complementary error function of X.
Domain of definition of these functions:

```
Erfc :        all values of X
Longerfc :    all values of X
```

```
real procedure Exp(real value X);
long real procedure Longexp(long real value X);
```

Returns the exponential of X.
Domain of definition of these functions:

```
Exp :        -180.2182 < X < 174.6730
Longexp :    -180.2182 < X < 174.6730
```

If underflow detection is off (the default) then argument values below -180.2182 will return a function value of zero. If underflows are being trapped, that is the UNFL predeclared reference has been been reassigned, arguments below this value will be treated as a run error.

```
integer procedure Exponent(real value X);
```

Returns the integer value of the unbiased hexadecimal exponent used in the computer's representation of a floating point number. Number representation is discussed in Appendix J. The return value is:

```
(1)  if X=0 then 0; otherwise
(2)  the largest integer I such that:
     I <= (Log(|X|) / Log(16)) + 1
```

```
integer procedure External(string(8) value Ename);
```

Returns the machine address of the external symbol definition name (ESDname) given by Ename. This ESDname must be given as a string constant. External allows an Algol W program to locate externally defined data areas or subroutines.

```
integer procedure Fullword(integer value N);
```

Returns a fullword representation of a halfword (2-byte) integer which is assumed to occupy the leading two bytes of N. The trailing two bytes are discarded. The operation is implemented as a 16-bit System/370 Shift Right Arithmetic operation.

```
real procedure Gamma(real value X);
long real procedure Longgamma(long real value X);
```

Returns the gamma function of X.
Domain of definition of these functions:

```
Gamma :        0.1381786'-75 < X < 57.57441
Longgamma :    0.1381786'-75 < X < 57.57441
```

```
integer procedure Halfword(integer value N);
```

Returns a halfword representation of a fullword (4-byte) integer given by N. After the operation a halfword representation of the number will occupy the leading two bytes of the result, with all bits zero in the trailing two bytes. The action is implemented as a 16-bit System/370 Shift Left Arithmetic operation. Domain of definition of this function: -32768 <= N <= 32767

```
complex procedure Imag(real value X);
long complex procedure Longimag(long complex value X);
```

Returns the complex value 0+Xi to the specified precision.

```
real procedure Imagpart(complex value Z);
long real procedure Longimagpart(long complex value Z);
```

Returns as a real value the imaginary component of the complex number Z to the specified precision.

```
string(12) procedure Intbase10(integer value N);
```

Returns a string encoding of N in radix 10 (decimal) form. The format is:

```
"b+DDDDDDDDDD"
```

where "b" is a blank; "+" is the sign (blank if N is zero or positive); and "D" is a decimal digit. Leading zeros are not suppressed.

```
string(12) procedure Intbase16(integer value N);
```

Returns a string encoding of N in radix 16 (hexadecimal), unsigned two's complement form. The format is:

"␣␣␣␣AAAAAAAA"

where "␣" is a blank; and "A" is a hexadecimal digit. Leading hexadecimal zeros are not suppressed.

```
integer procedure Link(string(32) value Proc_Name);
```

Returns the address of a dynamically built O/S Type I linkage subroutine which may call a main code Algol W procedure, subject to the following restrictions:

(1) Proc_Name specifies a string constant containing the name of the procedure to be called;
(2) The procedure so designated must be in scope at the point where the Link call is issued;
(3) Certain restrictions apply to the procedure parameters - see main text description in the section "External Linkages";
(4) Link may only be issued as a parameter to the Call predeclared procedure.

Link provides the mechanism whereby a called FORTRAN subroutine may call back during execution to invoke a main code Algol W procedure. Several widely available mathematical subroutine libraries, such as NAAS, require this facility.

```
real procedure Ln(real value X);
long real procedure Longln(long real value X);
```

Returns the natural logarithm (base e) of X.
Domain of definition of these functions:

```
Ln :        X > 0.0
Longln :    X > 0.0
```

```
real procedure Lngamma(real value X);
long real procedure Longlngamma(long real value X);
```

Returns the natural logarithm (base e) of the gamma function of X.
Domain of definition of these functions:

```
Lngamma :        0.0 < X < 4.293705'+73
Longlngamma :    0.0 < X < 4.293705'+73
```

```
real procedure Log(real value X);
long real procedure Longlog(long real value X);
```

Returns the logarithm to the base 10 of X.
Domain of definition of these functions:

```
Log :        X > 0.0
Longlog :    X > 0.0
```

```
integer procedure Number(bits value B);
```

Returns the integer with the two's complement representation given by B. Number representation is discussed in Appendix J.

```
logical procedure Odd(integer value N);
```

Returns the logical value: (N rem 2) = 1

```
real procedure Realpart(complex value Z);
long real procedure Longrealpart(long complex value Z);
```

Returns as a real value the real component of the complex quantity Z to the specified precision.

```
integer procedure Round(real value X);
```

Returns the value of the integer expression:

```
if X < 0
   then Truncate(X - 0.5)
   else Truncate(X + 0.5);
```

See the description of Truncate; the effect is to return the integer value nearest to X.

```
real procedure Roundtoreal(long real value X);
```

Returns the properly rounded real (short precision) value of the long precision value X.

```
real procedure Sin(real value X);
long real procedure Longsin(long real value X);
```

Returns the sine of X.
Domain of definition of these functions:

```
Sin :        |X| < (2**18)*pi = 823549.625
Longsin :    |X| < (2**50)*pi = 3.537118'+15
```

```
real procedure Sinh(real value X);
long real procedure Longsinh(long real value X);
```

Returns the hyperbolic sine of X.
Domain of definition of these functions:

```
Sinh :        |X| < 175.3662
Longsinh :    |X| < 175.3662
```

```
real procedure Sqrt(real value X);
long real procedure Longsqrt(long real value X);
```

Returns the square root of X.
Domain of definition of these functions:

```
Sqrt :        X >= 0.0
Longsqrt :    X >= 0.0
```

```
real procedure Tan(real value X);
long real procedure Longtan(long real value X);
```

Returns the tangent of X.
Domain of definition of these functions:

```
Tan :         |X| < (2**18)*pi = 823549.625
Longtan :     |X| < (2**50)*pi = 3.537118'+15
```

Both functions exclude the singularities at:

```
X = (n+1/2)*pi, where n = ..., -1, 0, 1, ...
```

```
real procedure Tanh(real value X);
long real procedure Longtanh(long real value X);
```

Returns the hyperbolic tangent of X.
Domain of definition of these functions:

```
Tanh :        all values of X
Longtanh :    all values of X
```

```
integer procedure Time(integer value N);
```

Returns an integer value determined by interrogation of the computer's clock. The time value returned, and its units, are determined by N:

```
N = -2     Elapsed time (1/60 second)
N = -1     Time of Day (1/60 second)
N = 0      Total CPU Time (1/100 minute)
N = 1      Total CPU Time (1/60 second)
N = 2      Total CPU Time (1/38400 second)
N = 3      Problem CPU Time (1/38400 second)
N = 4      Supervisor CPU Time (1/38400 second)
```

The result for any other value of N is undefined.

```
integer procedure Truncate(integer value X);
```

Returns the integer value I such that:

```
|I| <= |X| < |I| + 1 and I * X >= 0;
```

# APPENDIX G: PREDECLARED VARIABLES

This appendix gives a summary of all predeclared variables defined within the Algol W environment. The list is in alphabetical order of variable name. All of these variables may be assumed to be defined, with the relevant initial assignment statements, in an outer block enclosing the user source program. They may be redeclared within the user program block structure.

All variables listed here are fully described in the main manual. An initial value given as 'sysxcp' indicates a special system reference which is not directly accessible to user programs.

integer A_Count

Initial value = 1
Contains the value of the assertion counter maintained by the Algol W system. It is incremented by one for each Assert statement successfully completed, and printed in the error message if the assertion fails.

logical Attnmark

Initial value = false
If attention trapping has been enabled, using the Attntrap predeclared procedure, then this variable is set to 'true' when an attention interrupt occurs.

logical Canreply

This variable is initialized to 'true' if the executing program is being run at a conversational terminal, or 'false' if the program is being run in batch.

reference(Exception) Divzero

Initial value = sysxcp
This reference controls processing of the exceptional condition recognized when floating point division by zero is attempted.

reference(Exception) Endfile

Initial value = sysxcp
This reference controls processing of the exceptional condition recognized when an attempt to read from a file-or-device returns end-of-file. It does not affect end-of-file conditions returned by

Getcard or record absent conditions returned by Xgetcard; in these cases control always returns to the program.

real Epsilon

Initial value = 9.536743'-07
This variable contains the largest positive real number e provided by the implementation such that: 1 + e = 1

bits Error

The initial value of this variable is such that the Algol W input/output routines recognize it in context as a reference to the ERROR stream (MTS SERCOM).

record Exception

A predeclared record class with the declaration:

```
record Exception(
   logical Xcpnoted;
   integer Xcplimit;
   integer Xcpaction;
   logical Xcpmark;
   string(64) Xcpmsg);
```

Prior record assignments to the predeclared reference(Exception) variables Divzero, Endfile, Function, Intdivzero, Intovfl, Ovfl and Unfl will modify the error processor action when the relevant exceptional condition is recognized by Algol W.

logical Filemark

Initial value = false
This variable is set after every input operation. It becomes 'true' if an end-of-file condition is recognized. For the procedures Get, Geton, Read, Readon and Readcard, inspection of the Endfile reference would take control away from the program unless it had been previously reassigned. For Getcard and Xgetcard, inspection of the value of Filemark is the only way to detect end-of-file or record absent conditions.

long real Fn_Value

Initial value = 0.0L
This variable may supply a replacement returned value for an analytic predeclared function which has been called with an illegal argument. For this to occur there must have been an assignment of the predeclared reference Function with the Xcpaction field set to 2.

reference(Exception) Function

Initial value = sysxcp
This variable controls processing of the exceptional conditions recognized when a predeclared function call has an invalid argument. The function may not be defined for the argument; or the argument may be a singularity in the definition; or roundoff errors may prevent its computation.

integer I_W

Initial value = 14
This integer supplies the field width used in integer output where no format string is being obeyed (for example using Write). If I_W has too small a value to print the number, it will be overridden.

bits Input

The initial value of this variable is such that the Algol W input/output routines recognize it in context as a reference to the INPUT stream (MTS SCARDS).

reference(Exception) Intdivzero

Initial value = sysxcp
This reference controls processing of the exceptional condition recognized when integer division by zero is attempted.

reference(Exception) Intovfl

Initial value = sysxcp
This reference controls processing of the exceptional condition recognized when an operation resulting in integer overflow is attempted.

long real Longepsilon

Initial value = 2.22044604925031'-15L
This variable contains the largest positive long real number e provided by the implementation such that: 1L + e = 1L

bits Lowercase

The initial value of this variable is a pointer to a 256-byte translate table. Lowercase may be used directly as the second argument to the Translate predeclared procedure. It will cause all upper case alphabetic characters to be converted to their lower case equivalents leaving all other characters unchanged.

integer Maxinteger

Initial value = 2147483647
This variable contains the maximum positive integer which may be represented on the computer. See Appendix J for a discussion of number representation.

long real Maxreal

Initial value = 7.23700557733225'+75L
This variable contains the largest positive long real number which may be represented on the computer. See Appendix J for a discussion of number representation.

reference(Exception) Ovfl

Initial value = sysxcp
This reference controls processing of the exceptional condition recognized when an operation resulting in floating point exponent overflow is attempted.

long real Pi

Initial value = 3.14159265358979L
The ratio of the circumference to diameter of a circle.

bits Print

The initial value of this variable is such that the Algol W input/output routines recognize it in context as a reference to the PRINT stream (MTS SPRINT).

bits Punch

The initial value of this variable is such that the Algol W input/output routines recognize it in context as a reference to the PUNCH stream (MTS SPUNCH).

long complex R_Cmplx

Initial value = 0L+0IL
For complex or long complex functions accessed via the Call or Rcall predeclared procedures, the result will be found in this variable on return. The long real part of this variable shares its storage location with R_Float.

integer R_Code

Initial value = 0
External O/S Type I subroutines frequently leave an indication of success or failure in machine general register fifteen. This value, called a return code, will be found in R_Code on return. This applies to subroutines accessed via the predeclared procedures

Call or Rcall, or the obsolete FORTRAN linkage. R_Code is also used to supply a return code to a routine that has called an Algol W procedure using th O/S Type I calling convention. Algol W automatically places the value of R_Code in general register fifteen before returning to such a routine.

integer R_D

Initial value = 0
This integer supplies the decimal digit field width used for real, long real, complex, and long complex output where no format string is being obeyed (for example using Write). The value of R_D is only effective for values of R_Format which are "F" or "A".

string(1) R_Expchar

Initial value = "'"
This string specifies the exponent separator character used when real, long real, complex, or long complex values are output in explicit exponent form (for example 5.7'+05).

long real R_Float

Initial value = 0L
For real or long real functions accessed via the Call or Rcall predeclared procedures, the result will be found in this variable on return. This variable shares its storage location with the long real part of R_Cmplx.

string(1) R_Format

Initial value = "G"
This string specifies the format to be used in the output of real, long real, complex, or long complex values where no format string is being obeyed (for example using Write). Legal values are:

```
"A"       as for "F"
"D"       as for "E"
"E"       explicit exponent form:      5.935'+02
"F"       fixed decimal point form:        593.5
"G"       general form: "F" if possible otherwise "E"
"S"       as for "E"
```

Lower case may be supplied in place of the upper case values shown. All other values are treated as "G".

integer R_Sig

Initial value = 3
For R_Format value "G", this integer specifies the minimum number of significant digits which may be printed if the output is in "F" form. If less than this would be printed, the number is output in "E" explicit exponent form.

integer R_W

Initial value = 14
This integer supplies the field width used in real, long real, complex, and long complex output where no format string is being obeyed (for example using Write). If R_W has too small a value to print the number, it will be overridden.

bits Rdr

The initial value of this variable is such that the Algol W input/output routines recognize it in context as a reference to the basic input stream. This is initially INPUT (MTS SCARDS) but may be changed by a call of the Reader predeclared procedure.

integer R0

Initial value = 0
For integer or logical functions accessed via the Call or Rcall predeclared procedures, the value will be found in this variable on return. Additionally, Rcall will load general register zero from R0 before calling the subroutine. This variable shares its storage location with R01(0|4).

string(8) R01

Initial value = 8 bytes, all set to: code(0)
This variable allows the predeclared variables R0 and R1 to be accessed as strings. R01(0|4) has the same storage location as R0, and R01(4|4) shares the same location as as R1.

integer R1

Initial value = 0
For subroutines which set machine general register one and which are accessed via the Call or Rcall predeclared procedures, the value will be found in this variable on return. Additionally, Rcall will load general register one from R1 before calling the subroutine. This variable shares its storage location with R01(4|4).

integer S_W

Initial value = 2
This integer supplies the number of spaces to be appended when values of all simple types other than string are output where no format string is being obeyed (for example using Write).

integer Syscode

Initial value = 0
Where Algol W calls system subroutines in the course of obeying a predeclared function call, in certain cases the return code for such a subroutine is passed back in this variable. See the main manual for details.

integer Sysindex

Initial value = 0
Whenever Algol W reads a new physical input record or writes a new physical output record, the value of this variable is updated to reflect the record index (MTS line number times 1000) used in the operation.

string(256) Sysparm

Initial value = " " or "<parameter-string>" This variable is provided to allow a single record of input data to be supplied to an Algol W program as it is invoked. See the description under "Run Time Parameters" in the section "Algol W Programmer's Guide". See also the description of the DATAPARM parameter in the same section.

reference(Exception) Unfl

Initial value = null
This reference controls processing of the exceptional condition recognized when an operation resulting in floating point exponent underflow is attempted. Note the initial value; by default such conditions are ignored and will return a value of zero.

bits Uppercase

The initial value of this variable is a pointer to a 256-byte translate table. Uppercase may be used directly as the second argument to the Translate predeclared procedure. It will cause all lower case alphabetic characters to be converted to their upper case equivalents leaving all other characters unchanged.

bits User

The initial value of this variable is such that the Algol W input/output routines recognize it in context as a reference to the USER stream (MTS GUSER).

logical Write_Cc

Initial value = true or <CC/NOCC-setting>
This value of this variable controls the automatic generation of carriage control characters by the Write, Writeon and Writecard predeclared procedures. If 'true' such characters will be generated; this variable can be assigned within the program or by issuing the run-time parameters CC or NOCC.

bits Wtr

The initial value of this variable is such that the Algol W input/output routines recognize it in context as a reference to the basic output stream. This is initially PRINT (MTS SPRINT) but may be changed by a call of the Writer predeclared procedure.

integer Xcpaction

The third field of the predeclared exception record. This specifies the action to be taken to recover from an exceptional condition when program execution is to be resumed. Values other than zero, one or two are treated as zero.

integer Xcplimit

The second field of the predeclared exception record. This specifies the number of exceptions to be tolerated before a fatal run error condition is recognized and program execution stopped. It is decremented by one each time an exception is processed and program execution is resumed.

logical Xcpmark

The fourth field of the predeclared exception record. This specifies whether an error message is to be printed if an exception occurs. If program execution will not resume (fatal condition, Xcplimit < 1) an error message is always printed.

string(64) Xcpmsg

The fifth field of the predeclared exception record. If assigned and non-blank, this string is printed as part of the system error message when an exception is recognized. Printing or suppression of such messages is under the control of Xcpmark.

logical Xcpnoted

The first field of the predeclared exception record. When an exceptional condition is recognized, this field is set to 'true'.

# APPENDIX H: USER ORIENTED ALGOL W SYNTAX

The syntax given in this appendix is not complete. It is designed to aid the user already familiar with Algol W in determining the format of Algol W constructs. It is assumed that the user knows which variable and expression types are legal substitutes for <variable> and <expression>. For the complete Algol W syntax, see Appendix I.

```
<program>                ::= <statement> .       (notice the period)

<statement>              ::= <simple-statement>
                           |  <conditional-statement>
                           |  <iterative-statement>

<simple-statement>       ::= <assignment>
                           |  <block>
                           |  <empty-statement>
                           |  <assert-statement>
                           |  <procedure-statement>
                           |  <predeclared-procedure-statement>
                           |  <goto-statement>

<conditional-statement>  ::= <if-statement>
                           |  <case-statement>

<iterative-statement>    ::= <while-statement>
                           |  <for-statement>

<assignment>             ::= <variable> := [ <variable> := ]*
                               <expression>

<block>                  ::= begin [ <declaration> ; ]* <statement>
                               [ ; <statement> ]* end

<empty-statement>        ::=                              (nothing)

<assert-statement>       ::= assert <expression>

<procedure-statement>    ::= <identifier> [ ( <expression>
                               [ , <expression> ]* ) ]

<identifier>             ::= <letter>
                           |  <identifier> <letter>
                           |  <identifier> <digit>
                           |  <identifier> _
```

```
<letter>                ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|
                            T|U|V|W|X|Y|Z|
                            a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|
                            t|u|v|w|x|y|z
```

Note: Each lowercase alphabetic character is taken as the exact equivalent of the corresponding uppercase character.

```
<digit>                 ::= 0|1|2|3|4|5|6|7|8|9


<predeclared-procedure-statement> ::=
                            <predeclared-procedure-identifier> (
                            <expression> [ , <expression> ]* )
```

Note: The exact behavior of a particular predeclared procedure depends on the implementation, not the grammar. See the relevant section of the manual.

```
<predeclared-procedure-identifier> ::=
                              Read  |  Readon  |  Readcard
                             |  Write  |  Writeon  |  Writecard
                             |  Get  |  Geton  |  Put  |  Puton
                            | Getstring | Putstring | Getcard | Putcard
                             | Xgetcard | Xputcard | Xdelete | Reader
                             | Writer | Iocontrol | Newline | Release
                             |  Protect  |  Rewind  |  Empty
                             | Flush | Attntrap | Assign
                             | Qualify  |  Control  |  Sense | Trace
                             | Call | Rcall | Move | Fetch | Store
                             | Locate | Translate | Cmd | Stop

<variable>              ::= <simple-variable>
                             |  <subscripted-variable>
                             |  <field-designator>
                             |  <substring-designator>

<simple-variable>       ::= <identifier>

<subscripted-variable>  ::= <identifier> ( <expression>
                                [ , <expression> ]* )

<field-designator>      ::= <identifier> (<variable>)

<substring-designator>  ::=
                             <identifier> ( <expression> | <constant> )
```

Note: In the previous expression the "|" stands for itself. Also the double slash symbol "//" may be used as an alternative to the bar.

```
<expression>            ::= <operand> [ <operator> <operand> ]*
                          |  <if-expression>
                          |  <case-expression>

<operand>               ::= [ <unary-operator> ]*  <term>

<term>                  ::= <variable>
                          |  <constant>
                          |  <function-call>
                          |  <block-expression>
                          |  ( <expression> )

<operator>              ::= or  |  and
                          |  <   |  <=  |  >   |  >=  |  =   |  ¬=
                          |   +    |  -   |  *   |  /   | div  | rem
                          |  **  | shl  | shr

<unary-operator>        ::=  ¬  |  +   |  -
                          | abs  | long | short
```

Note: The reserved word 'not' is everywhere equivalent to the not symbol, ¬. The precedence of operators (highest to lowest) is:

```
long  short  abs
**  shl  shr
*  /  div  rem
+  -
<  <=  >  >=  =  ¬=
¬
and
or
```

```
<function-call>         ::= <identifier> [ ( <expression>
                                [ , <expression> ]* ) ]

<predeclared-function-call> ::=
                         <predeclared-function-identifier> (
                             <expression> )

<predeclared-function-identifier> ::=
                          External | Link | Odd | Bitstring | Number
                        | Decode | Code | Truncate | Round | Entier
                        | Exponent | Roundtoreal | Realpart
                        | Imagpart | Longrealpart | Longimagpart
                        | Imag | Longimag | Sqrt | Exp | Ln | Log
                        | Sin | Cos | Arctan | Tan | Cot | Arcsin
                        | Arccos | Sinh | Cosh | Tanh | Erf | Erfc
                        | Gamma | Lngamma | Longsqrt | Longexp
                        | Longln | Longlog | Longsin | Longcos
                        | Longarctan | Longtan | Longcot | Longarcsin
                        | Longarccos | Longsinh | Longcosh | Longtanh
                        | Longerf | Longerfc | Longgamma
                        | Longlngamma | Cxsin | Cxcos | Cxsqrt | Cxln
```

```
                        | Cxexp | Longcxsin | Longcxcos | Longcxsqrt
                        | Longcxln | Longcxexp | Intbase10
                        | Intbase16 | Base10 | Base16 | Longbase10
                        | Longbase16 | Date | Time | Halfword
                        | Fullword

<block-expression>      ::= begin [ <declaration> ; ]*
                            [ <statement> ; ]* <expression> end

<if-statement>          ::= if <expression> then <statement>
                          |  if <expression> then <simple-statement>
                            else <statement>

<if-expression>         ::= if <expression> then <expression>
                            else <expression>

<case-statement>        ::= case <expression> of begin <statement>
                            [ ; <statement> ]* end

<case-expression>       ::= case <expression> of ( <expression>
                            [ , <expression> ]* )

<while-statement>       ::= while <expression> do <statement>

<for-statement>         ::= for <identifier> := <expression>
                           [ step <expression> ] until <expression>
                            do <statement>
                          |  for <identifier> := <expression>
                            [ , <expression> ]* do <statement>

<declaration>           ::= <simple-variable-declaration>
                          |  <array-declaration>
                          |  <procedure-declaration>
                          |  <record-class-declaration>

<simple-variable-declaration> ::=
                            <type> <identifier> [ , <identifier> ]*

<type>                  ::= integer
                          | real
                          | long real
                          | complex
                          | long complex
                          | logical
                          | bits [ (32) ]
                          | string [ (<integer-constant>) ]
                          | reference (<identifier> [,
                              <identifier>]*)

<record-class-declaration> ::=
                           record <identifier>
                           ( <simple-variable-declaration>
                           [ ; <simple-variable-declaration>]* )
```

```
<array-declaration>     ::= <type> array <identifier>
                            [ , <identifier> ]* <bound pairs>

<bound-pairs>           ::= ( <expression> :: <expression>
                            [, <expression> :: <expression> ]* )

<procedure-declaration> ::= procedure <identifier>
                             [ <formal-parameter-list>   ] ;
                             <statement>
                          |  <type> procedure <identifier>
                               [ <formal-parameter-list>     ];
                               <expression>

<formal-parameter-list> ::= ( <parameter-declaration>
                            [ ; <parameter-declaration> ]* )

<parameter-declaration> ::= <type> [ <access> ] <identifier>
                            [ , <identifier> ]*
                          |  <type> array <identifier>
                            [ , <identifier> ]* ( * [ , * ]*)

<access>                ::= value
                          | result
                          | value result
```

454 User Oriented Algol W Syntax

# APPENDIX I: COMPLETE ALGOL W SYNTAX

The sets VT (the set of all Algol W terminal symbols), VN (the set of Algol W nonterminal symbols), and P (the set of all productions of Algol W) completely describe the language Algol W.

The sets VT and VN are defined through enumeration of their members below. The productions P are given throughout the remaining subsections. To provide explanations for the meaning of Algol W programs, lowercase letter sequences used as nonterminal symbols have been chosen to be English words describing approximately the nature of the syntactic entity or construct. Where words which have appeared in this manner are used elsewhere in the text, they refer to the corresponding syntactic definition. As a notational shorthand, along with these letter sequences the symbol T or Tn, where "n" is a digit, may occur. This symbol must be replaced by any one of a finite set of English words (or word pairs) given in the following table:

| | |
|---|---|
| integer | logical |
| real | bit |
| long real | string |
| complex | reference |
| long complex | |

If specifically stated, the replacement may be from a subset of the above. For example, the production

```
<T-expression-1> ::= <T-expression-2>
```

corresponds to

```
<integer-expression-1>      ::= <integer-expression-2>
<real-expression-1>         ::= <real-expression-2>
<long-real-expression-1>    ::= <long-real-expression-2>
<complex-expression-1>      ::= <complex-expression-2>
<long-complex-expression-1> ::= <long-complex-expression-2>
```

The production

```
<T4-expression-8> ::= long <T5-expression-8>
```

corresponds to

```
<long-real-expression-8>    ::= long <real-expression-8>
<long-real-expression-8>    ::= long <integer-expression-8>
<long-complex-expression-8> ::= long <complex-expression-8>
```

The symbol vertical bar "|" is an Algol W terminal symbol, but also a symbol of the metalanguage. To distinguish between the two, the Algol W vertical bar will be designated by <bar>.

The basic symbols are:

```
A  B  C  D  E  F  G  H  I  J  K  L  M
N  O  P  Q  R  S  T  U  V  W  X  Y  Z
a  b  c  d  e  f  g  h  i  j  k  l  m
n  o  p  q  r  s  t  u  v  w  x  y  z
0  1  2  3  4  5  6  7  8  9

true    false    "    null    #    '    <bar>    //
integer    real    complex    logical    bits    string
reference    array    procedure    record
,    ;    :    .    (    )    begin    end    if    then    else
case    of    +    -    *    /    **    div    rem    shr  shl  is
abs    long    short    and    or    ¬    not    _    =    ¬=    not=
<    <=    >    >=    ::
:=    goto    go to    for    step    until    do    while
comment    %    value    result    assert    algol    fortran
```

The symbol <empty> also is considered an element of the Algol W terminal symbol set, but it does not represent a physical symbol.

Note that the lower case letters are treated as if the upper case equivalent had been entered.

The following symbols are equivalent:

```
<bar>        //
¬            not
¬=           not=
```

In the descriptions which follow, only the first symbol of these pairs appears. It should be clearly understood that everywhere that the first symbol of one these pairs appears, it may always be replaced by the alternative symbol shown. The extra symbols were added because of the difficulty or confusion which can arise when entering "|" or "¬" from certain kinds of ASCII conversational terminals.

An index to the syntactic entities follows:

# 1 IDENTIFIERS

```
<identifier> ::= <letter> | <identifier> <letter> |
     <identifier> <digit> | <identifier> _
<T-variable-identifier> ::= <identifier>
<T-array-identifier> ::= <identifier>
<procedure-identifier> ::= <identifier>
<T-function-identifier> ::= <identifier>
<record-class-identifier> ::= <identifier>
<T-field-identifier> ::= <identifier>
<label-identifier> ::= <identifier>
<control-identifier> ::= <identifier>
<letter> ::= A | B | C | D | E | F | G | H | I | J | K | L | M |
             N | O | P | Q | R | S | T | U | V | W | X | Y | Z |
             a | b | c | d | e | f | g | h | i | j | k | l | m |
             n | o | p | q | r | s | t | u | v | w | x | y | z
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<identifier-list> ::= <identifier> |
     <identifier list> , <identifier>
```

# 2 VALUES

## 2.1 Numbers

```
<long-complex-constant> ::= <complex-constant>L
<complex-constant> ::= <imaginary-constant>
<imaginary-constant> ::= <real-constant>I |
     <integer-constant>I
<long-real-constant> ::= <real-constant>L |
     <integer-constant>L
<real-constant> ::= <unscaled-real> |
     <unscaled-real><scale-factor> |
     <integer-constant><scale-factor> |
     <scale-factor>
<unscaled-real> ::= <integer-constant>.<integer-constant> |
     .<integer-constant> | <integer-constant>.
<scale-factor> ::= '<integer-constant> |
     '<sign><integer-constant>
<integer-constant> ::= <digit> | <integer-constant><digit>
<sign> ::= + | -
```

## 2.2 Logical Values

```
<logical-constant> ::=  true | false
```

## 2.3 Bits Sequences

```
<bits-constant> ::= # <hex-digit> |
     <bits-constant><hex-digit>
<hex-digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
                | A | B | C | D | E | F
                | a | b | c | d | e | f
```

Note that the <hex-digits> 0 | ... | F correspond to the decimal numbers 0 | ... | 15 .

## 2.4 Strings

```
<string-constant> ::= <string> | <string> <string>
<string> ::= "<open-string>"
<open-string> ::= <character> | <open-string><character>
```

## 2.5 References

```
<reference-constant> ::= null
```

# 3 DECLARATIONS

```
<declaration> ::= <simple-T-variable-declaration> |
     <T-array-declaration> | <procedure-declaration> |
     <record-class-declaration>
```

## 3.1 Simple Variable Declarations

```
<simple-T-variable-declaration> ::= <T-type><identifier-list>
<integer-type> ::= integer
<real-type> ::= real
<long-real-type> ::= long real
<complex-type> ::= complex
<long-complex-type> ::= long complex
<logical-type> ::= logical
<bits-type> ::= bits | bits (32)
<string-type> ::= string | string (<integer-constant>)
<reference-type> ::=
     reference (<record-class-identifier-list>)
<record-class-identifier-list> ::= <record-class-identifier>|
     <record-class-identifier-list>,<record-class-identifier>
```

## 3.2 Array Declarations

```
<T-array-declaration> ::= <T-type> array <identifier-list>
     (<bound-pair-list>)
<bound-pair-list> ::=  <bound-pair> |
     <bound-pair-list>,<bound-pair>
<bound-pair> ::= <lower-bound> :: <upper-bound>
<lower-bound> ::= <integer-expression>
<upper-bound> ::= <integer-expression>
```

## 3.3 Procedure Declarations

```
<procedure-declaration> ::= <proper-procedure-declaration> |
     <T-function-procedure-declaration>
<proper-procedure-declaration> ::= procedure
     <procedure-heading>; <proper-procedure-body>
<T0-function-procedure-declaration> ::=
     <T0-type> procedure <procedure-heading>;
     <T1-function-procedure-body>
<proper-procedure-body> ::= <statement> |
     <external-reference>


<T-function-procedure-body> ::=  <T-expression> |
     <external-reference>
<procedure-heading> ::= <identifier> |
     <identifier> (<formal-parameter-list>)
<formal-parameter-list> ::= <formal-parameter-segment> |
     <formal-parameter-list>;<formal-parameter-segment>
<formal-parameter-segment> ::= <formal-array-parameter> |
     <formal-type><identifier-list>
<formal-type> ::= <T-type> | <T-type> value |
     <T-type> result | <T-type> value result |
     <T-type>  procedure | procedure
<formal-array-parameter> ::= <T-type>  array
     <identifier-list> (<dimension-specification>)
<dimension-specification> ::= * |
     <dimension-specification> , *
<external-reference> ::=  fortran <string> |  algol <string>
```

## 3.4 Record Class Declarations

```
<record-class-declaration> ::=
     record <identifier> (<field-list>)
<field-list> ::= <simple-T-variable-declaration> |
     <field-list>; <simple-T-variable-declaration>
```

## 4 EXPRESSIONS

Expressions are distinguished by a type and a precedence level, the former depending on the types of the operands and the latter resulting from the precedence hierarchy imposed upon operators in the syntactic rules which follow. The syntactic entities naming different kinds of expressions in these rules are chosen to illustrate their types and precedences, the word "expression" being prefixed by a type and, usually, postfixed by an integer indicating the precedence level. (Higher precedence is implied by increasing magnitude of this integer.) The operators and their precedence levels are:

| Level | Operators |
|---|---|
| 1 | or |
| 2 | and |
| 3 | ¬ |
| 4 | < <= = ¬= >= > is |
| 5 | + - |
| 6 | * / div rem |
| 7 | shl shr ** |
| 8 | long short abs |

When the types allow an operator at level "i" to be applied to operands, the resulting expression, which belongs to the syntactic class <T-expression-i>, has the hierarchical property given in the second column of the table below.

| Syntactic Entity | Property |
|---|---|
| <T-expression-1> | disjunction |
| <T-expression-2> | conjunction |
| <T-expression-3> | negation |
| <T-expression-4> | relation |
| <T-expression-5> | sum |
| <T-expression-6> | term |
| <T-expression-7> | factor |
| <T-expression-8> | primary |

Throughout this part, the symbol T has to be replaced uniformly as described above, and the triplets T0, T1, T2 have to be uniformly replaced by exactly one of the words:

logical
bit
string
reference

or (subject to specification to the contrary) in accordance with the following "triplet rules" (1) and (2):

(1) Given the attributes (integer, real, or complex) of T1 and T2, the corresponding attribute of T0 is given in the following table:

| | T2 | integer | real | complex |
|---|---|---|---|---|
| T1 | integer | integer | real | complex |
| | real | real | real | complex |
| | complex | complex | complex | complex |

(2) T0 has the quality "long" if either both T1 and T2 have that quality, or if one has the quality "long" and the other is "integer".

```
<T-expression> ::= <T-expression-1> |
      <conditional-T-expression>
<T-expression-1> ::= <T-expression-2>
<T-expression-2> ::= <T-expression-3>
<T-expression-3> ::= <T-expression-4>
<T-expression-4> ::= <T-expression-5>
<T-expression-5> ::= <T-expression-6>
<T-expression-6> ::= <T-expression-7>
<T-expression-7> ::= <T-expression-8>
<T-expression-8> ::= <T-variable> |
      <T-function-designator> | <T-constant> |
      (<T-expression>) | <T-block-expression>
<T-block-expression> ::= <block body><T-expression> end
```

## 4.1 Variables

```
<simple-T-variable> ::= <T-variable-identifier> |
     <T-field-designator> | <T-array-designator>
<T-variable> ::= <simple-T-variable>
<string-variable> ::= <substring-designator>
<T-field-designator> ::= <T-field-identifier>
     (<reference-expression>)
<T-array-designator> ::= <T-array-identifier>
     (<subscript-list>)
<subscript-list> ::= <subscript> |
     <subscript-list> , <subscript>
 <subscript> ::= <integer-expression>
```

## 4.2 Function Designators

```
<T-function-designator> ::= <T-function-identifier> |
     <T-function-identifier> (<actual-parameter-list>)
```

## 4.3 Arithmetic Expressions

```
<T3-expression-5> ::= + <T3-expression-6> |
     - <T3-expression-6>
<T0-expression-5> ::= <T1-expression-5> + <T2-expression-6> |
     <T1-expression-5> - <T2-expression-6>
<T0-expression-6> ::= <T1-expression-6> * <T2-expression-7> |
     <T1-expression-6> / <T2-expression-7>
<integer-expression-6> ::=
     <integer-expression-6> div <integer-expression-7> |
     <integer-expression-6> rem <integer-expression-7>
<T4-expression-7> ::=
     <T5-expression-7> ** <integer-expression-8>
<T4-expression-8> ::= abs <T5-expression-8> |
     long <T5-expression-8> | short <T5-expression-8>
<integer-expression-8> ::= <control-identifier>
```

## 4.4 Logical Expressions

In the following rules for <relation> the symbols T6 and T7 must either both be replaced by any one of the following words:

```
logical
bit
string
reference
```

or individually replaced by any of the words:

```
complex
long-complex
real
long-real
integer
```

and the symbols T8 or T9 must be simultaneously replaced by string or must be replaced by any of real, long-real, or integer.

```
<logical-expression-1> ::=
     <logical-expression-1> or <logical-expression-2>
<logical-expression-2> ::=
     <logical-expression-2> and <logical-expression-3>
<logical-expression-3> ::= ¬ <logical-expression-4>
<logical-expression-4> ::= <relation> | <logical-variable>
<relation> ::=
     <T6-expression-5><equality-operator><T7-expression-5> |
     <T8-expression-5><inequality-operator><T9-expression-5> |
     <reference-expression-5> is <record-class-identifier>
<equality-operator> ::= = | ¬=
<inequality-operator> ::= < | <= | >= | >
```

## 4.5 Bits Expressions

```
<bits-expression-1> ::=
     <bits-expression-1> or <bits-expression-2>
<bits-expression-2> ::=
     <bits-expression-2> and <bits-expression-3>
<bits-expression-3> ::= ¬ <bits-expression-4>
<bits-expression-7> ::=
     <bits-expression-7> shl <integer-expression-8> |
     <bits-expression-7> shr <integer-expression-8>
```

## 4.6 String Expressions

```
<substring-designator> ::= <string-variable>
     (<integer-expression><bar><integer-constant>)
```

## 4.7 Reference Expressions

```
<reference-expression-8> ::= <record-designator>
<record-designator> ::= <record-class-identifier> |
     <record-class-identifier> (<expression-list>)
<expression-list> ::= <empty> | <T-expression> |
     <expression-list>, |
     <expression-list> , <T-expression>
```

## 4.8 Conditional Expressions

```
<conditional-T-expression> ::=
     <case-clause> (<T-expression-list>)
<conditional-T0-expression> ::=
     <if-clause> <T1-expression> else <T2-expression>
<T-expression-list> ::= <T-expression>
<T0-expression-list> ::=
     <T1-expression-list> , <T2-expression>
<if-clause> ::= if <logical-expression> then
<case-clause> ::= case <integer-expression> of
```

## 5 STATEMENTS

```
<program> ::= <statement>. |
     <proper-procedure-declaration>. |
     <T-function-procedure-declaration>.
<statement> ::= <simple-statement> | <iterative-statement> |
     <if-statement> | <case-statement>
<simple-statement> ::= <block> | <T-assignment-statement> |
     <procedure-statement> | <goto-statement> |
     <predeclared-procedure-statement> |
     <assert-statement> | <empty>
```

### 5.1 Blocks

```
<block> ::= <block-body> <statement> end
<block-body> ::= <block-head> | <block-body> <statement> ; |
     <block-body> <label-definition>
<block-head> ::= begin | <block-head> <declaration>
<label-definition> ::= <identifier> :
```

### 5.2 Assignment Statements

In the following rules the symbols T0 and T1 must be replaced by words which may be substituted for T as indicated at the beginning of this appendix, subject to the restriction that the type T1 must be assignment compatible with the type T0.

```
<T0-assignment-statement> ::= <T0-left-part><T1-expression> |
     <T0-left-part><T1-assignment-statement>
<T-left-part> ::= <T-variable> :=
```

### 5.3 Procedure Statements

```
<procedure-statement> ::= <procedure-identifier> |
     <procedure-identifier> (<actual-parameter-list>)
<actual-parameter-list> ::= <actual-parameter> |
     <actual-parameter-list> , <actual-parameter>
<actual-parameter> ::= <T-expression> | <statement> |
     <T-subarray-designator> | <procedure-identifier> |
     <T-function-identifier>
<T-subarray-designator> ::= <T-array-identifier>  |
     <T-array-identifier>(<subarray-designator-list>)
<subarray-designator-list> ::= <subscript> | * |
     <subarray-designator-list>,<subscript> |
     <subarray-designator-list>,*
```

## 5.4 Goto Statements

```
<goto-statement> ::= goto <label-identifier> |
     go to <label-identifier>
```

## 5.5 If Statements

```
<if-statement> ::= <if-clause><statement> |
     <if-clause><simple-statement> else <statement>
<if-clause> := if <logical-expression> then
```

## 5.6 Case Statements

```
<case-statement> ::= <case-clause> begin <statement-list> end
<statement-list> ::= <statement> |
     <statement-list>;<statement>
<case-clause> ::= case <integer-expression> of
```

## 5.7 Iterative Statements

```
<iterative-statement> := <for-clause><statement> |
     <while-clause><statement>
<for-clause> ::= for <identifier> := <initial-value>
     step <increment> until <limit> do |
     for <identifier> := <initial-value> until <limit> do |
     for <identifier> := <for-list> do
<for-list> ::= <integer-expression> |
     <for-list> , <integer-expression>
<initial-value> ::= <integer-expression>
<increment> ::= <integer-expression>
<limit> ::= <integer-expression>
<while-clause> ::= while <logical-expression> do
```

## 5.8 Assert Statements

```
<assert statement> ::= assert <logical expression>
```

## 5.9 Predeclared Procedures

```
<predeclared-procedure-statement> ::=
     <predeclared-procedure-identifier>
          (<predeclared-procedure-parameter-list)
<predeclared-procedure-parameter-list> ::=
     <input-parameter-list>) | <transput-parameter-list>
<input-parameter-list> ::= <T-variable> |
     <simple-statement> |
     <input-parameter-list> , <T-variable> |
     <input-parameter-list> , <simple-statement>
<transput-parameter-list> ::= <T-expression> |
     <simple-statement> |
     <transput-parameter-list> , <T-expression> |
     <transput-parameter-list> , <simple-statement>
```

Note: The exact form of a predeclared procedure statement depends on the implementation of a particular predeclared procedure. See the main text description for the required procedure.

468  Complete Algol W syntax

# APPENDIX J: INTERNAL REPRESENTATION OF NUMERICAL DATA

On System/370 type computers (e.g. Amdahl 470 computers), the following units of storage are used:

(1) the bit; a single 0 or 1

(2) the byte; a group of eight consecutive bits

(3) the word; a group of four consecutive bytes; that is 32 consecutive bits

(4) the doubleword; a group of 2 consecutive words; that is eight bytes or 64 bits.

In practice the byte is the smallest unit of addressable data. It is the smallest unit of storage whose position in main storage may be specified directly.

For number representation in Algol W words and doublewords are the main units of interest.

## INTEGERS

Integers are stored in words. Of the 32 bits of a word, the leading one is reserved for the sign (0 for + and 1 for -, leaving 31 bits to represent the magnitude. A positive or zero valued integer is stored in a binary (base 2) representation. Thus:

$$21_{10}$$

(the subscript means base 10) is stored as:

```
0000  0000  0000  0000  0000  0000  0000  0001  0101
|
sign bit
```

To confirm this, note that:

$$0x2^{30} + \ldots + 0x2^{5} + 1x2^{4} + 0x2^{3} + 1x2^{2} + 0x2^{1} + 1x2^{0}$$

gives the decimal value 21.

The largest integer that can be stored in a word is:

$$2^{30} + 2^{29} + \ldots + 2^{1} + 2^{0} = 2^{31} - 1$$

which is:

$$2147483647_{10}$$

Any attempt to create or store an integer larger than this will produce erroneous results, and the user will usually, but not always, be warned of this error (see below).

To save space in writing words on paper, each group of four bits in a word is frequently converted to a single base 16 (hexadecimal) digit, according to the following code:

| base 2 | base 16 | base 2 | base 16 |
|---|---|---|---|
| 0000 | 0 | 1000 | 8 |
| 0001 | 1 | 1001 | 9 |
| 0010 | 2 | 1010 | A |
| 0011 | 3 | 1011 | B |
| 0100 | 4 | 1100 | C |
| 0101 | 5 | 1101 | D |
| 0110 | 6 | 1110 | E |
| 0111 | 7 | 1111 | F |

Thus the letters A, B, C, D, E and F are used as base 16 representations of the decimal numbers 10, 11, 12, 13, 14 and 15. Nevertheless, integers are stored as base 2 numbers.

Using hexadecimal notation, the decimal number 21 is written:

$$00000015_{16}$$

Note that:

$$15_{16} \text{ and } 21_{10}$$

are the same integer number in different representations.

Negative integer values are stored in what is called two's complement form. For example, the value -1 is stored as:

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111 = FFFFFFFF_{16}$$

Similarly, the value -21 is stored as:

1111 1111 1111 1111 1111 1111 1110 1011 = FFFFFFEB
16

The representation for -21 is obtained from that for +21 by changing every 0 to 1 and every 1 to 0, and then adding +1 in base 2 arithmetic to the result. Similarly for any other negative integer. Note that every negative integer has 1 as its sign bit. The largest negative integer value which may be stored in a word is:

$-2^{31} = -2147483648_{10}$

and it is represented by:

$80000000_{16}$

Another way to think of the representation of negative numbers is to consider a 32-place binary accumulating register. If one starts with all zeros in this register, the representation of -1 can be obtained by subtracting 1. The process requires a "borrow" to propogate to the left all the way across the register, leaving all ones. Continued subtraction will give the representation for -2, -3, and so on.

In an accumulator of this kind it can also be seen what happens when a positive number greater than 2147483647 is created. For example, if 1 is added to this number, the resulting carry will go all the way into the sign bit. This leaves a sign bit of 1, with all other bits zero -- which is the representation of -2147483648, the largest negative integer. Consequently positive integers that overflow in this way may be sensed as negative by the computer.

The mechanisms of Algol W for detecting integer overflow of this kind can be used to detect additions, subtractions or multiplications that produce an integer, n, outside the range:

$-2^{31} <= n <= 2^{31} - 1$

Attempts to divide an integer by zero will yield an error message and meaningless values for quotient and remainder.

If a user suspects that integers in a program are getting anywhere near the limit of representation, then conversion to double precision floating point by assignment to a long real variable should be considered. Conversion to single precision floating point numbers may lose some precision.

The most important thing for a scientific user to remember is that integers in the allowed range are stored without approximation. Moreover, operations on integers (addition, subtraction and multiplication) are done without any error, while all intermediate and final results stay within the allowed range.

It is perhaps easier to remember the safe range:

$$-2 \times 10^9 < n < 2 \times 10^9$$

which is obtained from the useful approximation that:

$$2^{10} = 10^3$$

The operations of division without remainder (called Div in Algol W) and taking the remainder on division (called Rem in Algol W) always give integer results. If the divisor is zero, an error message is given.

In Algol W two operations on integers give results that are not integer. They are / (divide) and ** (power).

## FLOATING POINT NUMBERS

Numbers in many scientific computations will grow in magnitude well beyond the range of integers described above. To provide for this, System/370, Amdahl 470 V/8, and most scientific computers have a second way to represent numbers. This is the so-called floating point representation. The significance of the name "floating point" is that the radix point -- for example, the decimal point in base 10 numbers -- is permitted to float to the right or left, thus permitting scaling of numbers by various powers of the radix.

Although a decimal point that has floated off to the left will produce a number like 0.001345, the numbers are actually represented in a form closer to what is often called scientific notation, here:

$$1.345 \times 10^{-3}$$

In System/370 type machines, floating point numbers are always represented in base 16 notation; that is the radix or number base is 16. This permits the writing of numbers in abbreviated form (as was done with integers earlier). More important, the use of base 16 conforms with hardware arithmetic processes in which shifting is done four bits at a time to speed up the operations. This increase in speed is achieved at a slight cost in precision, as is learned from detailed error analyses which would be out of place here.

Consider first the floating point representation of numbers by a single word of 32 bits. This is the so called single precision or short real number, the number of type real in Algol W. The 32 bits of a word are numbered from 0 to 31, from left to right, to aid in their identification.

In floating point representation the left hand eight bits (bits 0 to 7, equivalent to two hexadecimal digits) are devoted to the sign of the number and the exponent of 16 associated with the number. The right

hand 24 bits (bits 8 to 31, equivalent to six hexadecimal digits) represent six significant hexadecimal digits (the significand) of the number.

As with integers, the sign of the number is denoted by bit 0, with 0 representing + and 1 representing -.

Bits 1 to 7 give the binary (base 2) representation of a non-negative integer, e, in the range:

$$0_{10} \text{ to } 127_{10}$$

inclusive. This integer is called the biased exponent, for reasons now to be explained. If this integer were taken directly as the exponent, there could be no negative exponents, and therefore no way of representing such numbers as:

$$16^{-25}$$

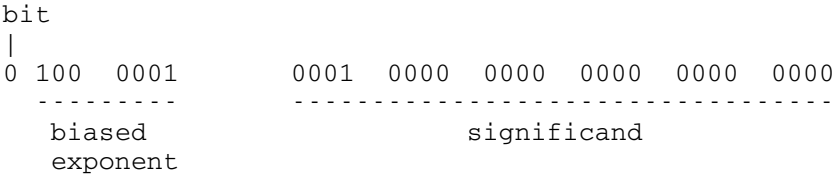
It is desirable to have an exponent range which is approximately symmetric about zero. In the computer the true exponent of a floating point number is obtained by subtracting decimal 64 from the biased exponent represented by bits 1 to 7. As a result, the actual exponents range from decimal -64 to +63.

The 24 bits 8 to 31 of a number are regarded as six hexadecimal digits with a hexadecimal point at the left hand end (that is before the digits). If the floating point number zero is being represented, all the hexadecimal digits are zero, as are all the other bits. Otherwise, at least one of the hexadecimal digits must be non-zero.

A floating point number is said to be normalized if the left most hexadecimal digit (the most significant digit) is non zero. In ordinary circumstances the computer's floating point numbers are normalized, and other forms will not be considered here.

Consider now the floating point representations of some sample numbers. As was previously stated, the number zero is represented by 32 zero bits, that is eight hexadecimal zero digits. Thus zero is represented by the same words in floating point or integer form. No other number has this property.

The number 1.0 is represented by:



To check this, note that the sign bit is zero representing plus. The biased exponent is:

$$1000001_{2} \text{ or } 65_{10}$$

Subtracting decimal 64 yields 1 as the true exponent. The hexadecimal significand is:

$$100000_{16}$$

Putting a hexadecimal point at the left end gives the hexadecimal fraction .100000, which equals 1/16 (one sixteenth). Thus the above word represents:

$$+ (1/16) \times 16^{1} \text{ or } 1.0_{10}$$

To save writing, the above word is usually written in the hexadecimal form 41100000 . While some numbers can be recognized in this form, there is in general no easy way to convert such a hexadecimal word into a real number. The number must be examined, and the six right most hexadecimal digits are prefixed by a hexadecimal point. If the remaining two left most digits are less than hexadecimal 80, then the number is positive and the true exponent is obtained by subtracting hexadecimal 40 (= decimal 64). If this two digit number is hexadecimal 80 or greater, then the number is negative and the true exponent is obtained by subtracting hexadecimal C0 (80 + 40, or decimal 192). In dealing with such numbers some facility with hexadecimal arithmetic is required.

In this presentation the radix point has been considered to be at the left of the six significant hexadecimal digits, with the exponent biased high by decimal 64. As an alternative, the radix point may be placed just to the right of the most significant digit of the significand, and the exponent may now be regarded as high by decimal 65. This brings the significand closer to the usual scientific notation but requires a trickier conversion to get the true exponent. The fact that either interpretation (and many others) are possible shows that really the radix point is just in the eye of the beholder, and not in the machine.

Here are some examples of floating point numbers, together with the computer's hexadecimal versions:

| Decimal number | machine floating point |
|---|---|
| 0.0 | 00000000 |
| 1.0 | 41100000 |
| 0.0625 | 40100000 |
| 16.0 | 42100000 |
| 256.0 | 43100000 |
| -1.0 | C1100000 |
| -16.0 | C2100000 |
| 3.5 | 41380000 |

The largest floating point number is 7FFFFFFF, representing:

$$.FFFFFF \times 16^{3F} \quad \text{or} \quad (1 - 16^{-6}) \times 16^{63} \quad \text{or about} \quad 7.23 \times 10^{75}$$

(Here 10 and sixteen denote decimal numbers.)

The smallest positive normalized floating point number is 00100000, representing:

$$(1/16) \times 16^{-64} \quad \text{or about} \quad 5.40 \times 10^{-79}$$

The negatives of these two numbers can also be represented (merely by setting the sign bit), and they are the extremes of representable negative numbers.

Very few numbers can be exactly represented with six significant decimal digits. For example, one third is .333333 decimal only approximately. In the same way very few numbers can be exactly represented with six significant hexadecimal digits. For example, one third is .555555 hexadecimal only approximately. Moreover, some numbers that are exactly representable in decimal are only approximately representable in hexadecimal. For example, one tenth is .100000 decimal exactly, but .19999A hexadecimal only approximately.

Thus round-off error enters into the representation of most floating point numbers on computers, and the round off differs from that with decimal numbers. This can easily give rise to unexpected results. For example, if the above number .19999A hexadecimal (decimal .1) is multiplied by the integer decimal 100 (64 in hexadecimal), the result is not A.00000 hexadecimal (the expected 10.0 decimal), but instead A.00003 hexadecimal, as a cumulative effect of the slightly high approximation to .1 decimal. The value A.00003 hexadecimal rounds to 10.00002 on conversion to decimal.

The precision of a single precision hexadecimal number is roughly 1.0'-7 in Algol W notation. This can be thought of a being crudely equivalent to seven significant decimal digits.

Not only do errors appear in the representation of numbers inside computers, but they arise from arithmetic operations performed on

numbers. For example, the product of two floating point numbers may have up to 12 significant hexadecimal digits. When the product is stored as a single precision floating point number, it must be rounded to six hexadecimal digits. This introduces an error, even though the factors may have been exact.

When an Algol W program assigns decimal numbers or integer values to variables of type real, these are immediately converted to hexadecimal floating point numbers, with (usually) a round off error. When a program outputs numbers from the computer in Algol W, they are converted to decimal. Both conversions are done as well as possible, but introduce changes in the numbers that a programmer should be aware of. Of course, all intermediate operations introduce further round off and possible errors. It is unthinkable to do the analysis necessary to counteract these errors and get the true answer to the problem. If answers entirely uncontaminated by round off are required, integer arithmetic must be used with the requisite guards against overflow.

Fortunately most users can accept an indeterminate amount of round off in their numbers, provided that they have some assurance that round off is not growing out of control. It is the business of numerical analysts to provide algorithms whose round off properties are reasonably under control, and such work produces standard libraries of procedures such as the NAAS subroutine library.

## DOUBLE PRECISION

The precision of single precision floating point numbers is not adequate for most scientific and engineering purposes. This is because a considerable number of computations require still more precision at intermediate stages, just in order to retain normal accuracy at the end. As a result, System/370 type computers provide an easy mechanism for getting a great deal more precision. For this purpose a doubleword of 64 bits is used to store a floating point number of so called double precision or long precision.

In this representation, the sign and biased exponent are found in the first byte of the doubleword, with precisely the same interpretation as with single precision floating point numbers. The second word of the doubleword consists of eight hexadecimal digits immediately following the six found in the first word. There is no sign or exponent in the second word. Thus a doubleword represents a signed floating hexadecimal number with 14 significant hexadecimal digits. As before, non zero numbers are normalized so that the most significant (left most) digit of the 14 is non zero.

Examples:

```
1.0L      =  41 100000   00000000
0.1L      =  40 199999   9999999A
                ----------------
              long significand
```

There is a full set of arithmetic operations for both single and double precision operations. Single precision arithmetic operations take slightly more CPU time than that for double precision operations. For modest problems the extra time is completely dwarfed by the time lost to system operations, and the use of double precision is strongly recommended for all scientific computation. The only disadvantage of using long precision is the doubling of storage needed. If arrays with tens of thousands of elements exist, the extra storage may be costly, though with modern machines this is not so important as it used to be.

Since:

$16^{-14}$ is about $10^{-17}$

the double precision numbers are crudely equivalent in precision to 17 significant decimal digits.

For a machine with the speed of System/370 or Amdahl 470 machines, a number precision of six hexadecimal digits (roughly seven decimals) is considered very low, while a precision of 14 hexadecimal digits (roughly 17 decimals) is much more useful.

The floating point arithmetic hardware of System/370 type machines provides the possibility of detecting when numbers have gone outside the exponent range stated above. It may be thought that a range from roughly decimal exponent -79 to +75 should cover all reasonable computations. While exponent overflow and exponent overflow are not very common, they can be the cause of elusive errors. The evaluation of a determinant is a common computation, and for a matrix of order 40 is quite rapidly done. If the matrix elements are of the quite reasonable magnitude 1.0'-3, then the magnitude of the determinant will be no larger than roughly 1.0'-90 (and probably much smaller), well below the range of representable floating point numbers. Such problems are a frequent source of exponent underflow.

The mechanisms within Algol W for detecting exponent overflow and underflow are discussed elsewhere in this manual. Even without these, floating point numbers behave well for numbers that are at least 1.0'66 times as large as the largest integer in the system. Hence use of floating point numbers solves most of the problems raised by integer overflow. This also permits the use of a large set of rational numbers, which do not even enter the integer system.

ALGOL W REAL AND LONG REAL

Information on how to represent real variables and numbers to take advantage of both single and double precision will be found in the section "Values and Types."

The purpose of this section is to bring this information into rapport with the hardware representation of numbers. If a variable X is declared real, one word is set aside for its values, and it will be stored in single precision form. If a variable Xx is declared to be long real, a doubleword is set aside to hold its values, and it will be stored in double precision form.

If the number is written in one of the decimal floating point forms either without a trailing "L" or without specifying LONG as a compilation parameter, then it will be truncated to single precision, no matter how many digits are set down. Thus 3.1415926535897932 will be immediately truncated to single precision in the program, and all the superfluous digits are lost at once. The assignment:

```
Xx := 3.1415926535897932
```

will result in the doubleword Xx receiving an approximation to that value in the more significant word, and all zeros in the less significant word. Thus a precision of only approximately seven decimals is obtained for the pain of writing 17, and this may well contaminate all the rest of the computation.

If Xx is required to be precise to approximately the full double precision, one must either code the statement in the form:

```
Xx := 3.1415926535897932L
```

or give the compilation parameter LONG. This parameter causes all real quantities in the program to be automatically treated as if they had been specified as long real. This applies to all declarations, numerical constants, and predeclared variables and functions. If this parameter is used, for safety it should be kept with the program source on a /COMPILE control record.

With the declaration 'real X', and no LONG parameter specified, the statement:

```
X := 3.1415926535897932L
```

will result in X having a single precision approximation to the number, as the long representation is truncated on assignment to X.

When integer numbers are assigned to a variable declared as long real, the full accuracy of the number is retained, as the integer is floated to the long representation on assignment. The assignment:

```
Xx := 2147483647
```

will not lose accuracy even though no trailing "L" has been given, because the constant on the right hand side is an integer, and all integers possible on the machine may be represented exactly in double precision floating point form.

The previous information refers to numbers written within Algol W source programs. When floating point values are read in to variables using one of the predeclared procedures provided for that purpose, the conversion is always done to double precision accuracy. No trailing "L" is required, and its presence is flagged as illegal. If the destination variable is of single precision, then the double precision number converted from the character stream input is then truncated to single precision.

Certain aspects of the above may have unexpected effects when combined with the rules governing evaluation of arithmetic expressions. Assume the following declarations:

```
real X, Y, Z;
long real Xx, Yy, Zz;
integer I, J, K;
```

Then X*Y, I**J, and I*X are all long real. The assignment statement:

```
Xx := X := Y*Z
```

will result in Xx having a single precision truncated version of Y*Z in the more significant word, and zeros in the less significant word.

Moreover, I*I is integer, but I**2 is long real.

If a programmer understands the language Algol W and the preceding pages on number representation, then the effects of mathematical algorithms should be understandable. A certain caution regarding the effect of a computer on numbers entrusted to it is never out of place.

# APPENDIX K: SUBROUTINE CALLING CONVENTIONS

Some knowledge of System/370 assembly language is required before attempting to read this appendix.

## INTRODUCTION

A calling convention is a very rigid specification of the sequence of instructions to be used by a program to transfer control to another program (usually referred to as a subroutine). It is very desirable, although not always practical, to have only one set of conventions to be used by all programs no matter what language they are written in, so that FORTRAN programs may call assembly language programs and so forth. In MTS, the O/S Type I calling conventions have been adopted as the standard. A complete specification of these standards can be found in the IBM publication, "OS/360 System Supervisor Services and Macro Instructions", form number GC28-6646. This description will attempt to bring out the pertinent details of these calling conventions.

Throughout this discussion we will refer to the terms calling program, called program, save area, and calling sequence. The calling program is the program which is in control and wants to call another program (subroutine). The called program is the program (subroutine) which the calling program wants to call. The save area is an area belonging to the calling program which the called program uses to save and later restore general-purpose registers. The save area has a very rigid format and is discussed in more detail later on. A calling sequence is the actual sequence of machine instructions which perform the tasks as specified by the calling conventions.

The facilities that must be provided by the calling conventions are:

(1) Establish addressability and transfer to the entry point.

(2) Pass parameters on to the called program.

(3) Pass results back to the calling program.

(4) Save and restore general-purpose and floating-point registers.

(5) Reestablish addressability and return to the calling program.

(6) Pass a return code (error indication) back to the calling program so it knows how things went.

The remainder of this description will describe the O/S Type I calling conventions to show how they are used and how the facilities listed above are provided for.

## REGISTER AND STORAGE VARIANTS OF CALLS

The O/S Type I calling conventions actually consist of two very similar sets, referred to as S-type and R-type. The two differ only in the way parameters and results are passed between the calling and called programs. The R refers to register and the S to storage.

The R-type calling conventions utilize the general-purpose registers 0 and 1 for passing parameters and results. This allows only two parameters or results. The required calling sequence cannot be generated in many higher-level languages. Its advantages are that calling sequences are shorter and take less time to set up. These are very popular in lower-level system subroutines such as GETSPACE or GETFD. Algol W users needing to call subroutines that utilize R-type calling conventions can use the Rcall predeclared procedure described in this volume.

The S-type calling conventions require a pointer in register one to a vector of address constants called a parameter list. Since this list can be of any required length, several parameters can be passed using S-type calling convention. These conventions are used by system subroutines such as SCARDS or LOAD and are generated by all function or subprogram references in FORTRAN. Results can be passed back by giving variables in the parameter list new values or via register 0. Algol W users needing to call S-type subroutines can use the Call predeclared procedure described in this volume.

## PARAMETER LISTS

As stated above, a parameter list is a vector of address constants. The parameter list must be on a fullword boundary and the entries are each four bytes long. The address of the first parameter is the first word of the list, the address of the second parameter the second word of the list, and so on. For example, the parameter list for the Algol W statement:

```
Call("QQSV", X, Y, Z);
```

might be written in assembly code as:

```
PAR     DC    A(X)      address of X
        DC    A(Y)      address of Y
        DC    A(Z)      address of Z
```

Now this works well enough when the parameter list for the subroutine is of fixed length, but there is not enough information yet to allow a subroutine to determine the length of the parameter list and hence accept variable-length lists. For this reason there are two types of parameter lists, fixed-length as described above, and an extended form called a variable-length parameter list which is described next.

Since a standard System/370 type computer uses 24-bit storage addresses, the left-most byte of an address constant is usually zero. In a variable-length parameter list, bit zero of the left-most byte of the last parameter address constant is set to 1 to show that it is the last item in the list. The example above then would be written as:

```
PAR      DC    A(X)      address of X
         DC    A(Y)      address of Y
         DC    XL1'80'   turn on bit zero
         DC    AL3(Z)    address of Z
```

if it generated a variable-length parameter list, as Algol W does. work with a variable-length parameter list, provided that it is at least as long as the fixed-length list the program is expecting.Such a routine extracts only the address part when it uses the parameters.

## REGISTER ASSIGNMENTS

Of the sixteen general-purpose registers, five are assigned for use in the calling conventions. The use of the general registers differs slightly depending upon whether an R- or S-type call is being made. Table 1 specifies exactly what each register is used for during a call.

Notice that it is the called program's responsibility to save and restore registers 2-12 in the save area provided by the calling program. There are two reasons for this. First, only the called program knows how many of the registers from 2-12 it is going to use. Since a register need be saved and restored only if it is actually going to be changed, the called program may be able to save some time by saving and restoring only those registers which it will use. Secondly, the called program requires addressability over the area in which it will save registers upon entry, since any attempt to acquire the address of a save area would destroy some of the registers which are to be saved. Furthermore, the save area should not be a part of the called program since that would prevent it from being reentrant (shareable). This means the calling program should provide the save area in which registers are saved and restored. And so we have the called program saving and restoring registers 2-12 in a save area provided by the calling program.

The calling conventions are quite different with floating-point registers. Since a large percentage of programs do not leave items in floating-point registers across subroutine calls it seems rather waste-

ful to always save and restore the floating-point registers. So the convention has been established that the calling program must save and restore those floating-point registers that contain items which are wanted. Also, programs that return a single floating-point result quite frequently do so via floating-point register zero.

Table 1: General-Purpose Register Conventions

| Register Number | Contents |
|---|---|
| 0 | Parameter to be passed in R-type sequences.<br><br>Result to be passed back in R- and S-type sequences. |
| 1 | Parameter to be passed in R-type sequences.<br><br>Address of a parameter list in S-type sequences. |
| 2-12 | Not used as a part of the calling sequence. Must be saved and restored by the called program. The save area is used for this. |
| 13 | The address of the save area provided by the calling program to be used by the called program. |
| 14 | Address of the location in the calling program to which control should be returned after execution of the called program. |
| 15 | Address of the entry point in the called program at the time of the call.<br><br>A return code at the time of the return that indicates to the calling program whether or not an exceptional condition occurred during processing of the called program. The return code should be zero for a normal return or a multiple of four for various exceptional conditions. |

## RETURNING RESULTS

There are in the calling conventions four ways in which a subroutine can return a result. These are:

(1) Value of result in general-purpose register zero.

(2) Value of result in general-purpose register one.

(3) Value of a result in floating-point registers (usually F/P register zero).

(4) Value of a parameter from the parameter list changed.

The particular method used depends upon whether the R- or S-type convention is used and whether the called program is intended to be a function (in FORTRAN arithmetic statements).

The first three methods are used by R-type calling conventions for all returned results. The contents of each of the registers depends upon the particular called program and are described in the subroutine description for each subroutine using the R-type calling conventions.

The first, third, and fourth methods are used by S-type calling conventions for all returned results. The first and third methods are used by function subprograms whose calls can be embedded in FORTRAN statements. The choice of general register 0 or floating-point register 0 depends upon whether the result returned is integer or floating point mode, respectively. Examples of subroutines which return results in this manner are the FORTRAN IV Library Subprograms, such as EXP, ALOG, or SIN. The fourth method can be used by a subprogram. An example would be a subprogram called from Algol W by:

```
Call("MATADD", A(1,1), B(1,1), C(1,1), M, N);
```

which might add the MxN matrices A and B together and store the result in C.

## SAVE AREA FORMAT

The save area is an area belonging to the calling program which the called program uses to save and later restore general-purpose registers. The address of the save area is passed to the called program by the calling program via general-purpose register 13. The save area has a very rigid format and is described in Table 2.

There are two things to be noted about the save area format, namely, who sets what parts of the save area and how these areas might be set up. The calling program is responsible for setting up the second word of the save area. This is to contain the address of the save area which was provided when the calling program was called. Although this is technically set up by the calling program as a part of the call, most programs set up the save area they will provide to subroutines they call once and leave its address in general register 13. This process then does not need to be repeated for each call. The called program is responsible for setting up the third through eighteenth words of the save area. The called program usually saves the general registers which it will use as a part of its initialization procedure and restores the registers as a part of the return procedure. Notice that the save area

format is amenable to use of the store multiple and load multiple instructions for saving and restoring blocks of registers. All of this will be made clearer in the examples at the end of this section.

Some MTS system subroutines (notably GETSPACE, FREESPAC, and a few others) do not require that a save area be provided for them. For these subroutines general register 13 need not be set up before a call; its contents are preserved by the called subroutine. The subroutines which need no save area are clearly marked as such in the MTS subroutine descriptions. Note that it is all right to provide a save area to one of these subroutines; it will simply be ignored.

Table 2: Save Area Format

| Word | Displacement | Contents |
|---|---|---|
| 1 | 0 | Used by FORTRAN, PL/1, and other beasties for many devious purposes. Don't touch! |
| 2 | 4 | Address of the save area used by the calling program. Forms a backward chain of save areas. Stored by calling program. |
| 3 | 8 | Address of the save area provided by the called program for programs it calls. Forms a forward chain of save areas. |
| 4 | 12 | Return address. Contents of register 14 at time of call. |
| 5 | 16 | Entry point address. Contents of register 15 at time of call. |
| 6 | 20 | Register 0 contents. |
| 7 | 24 | Register 1 contents. |
| 8 | 28 | Register 2 contents. |
| 9 | 32 | Register 3 contents. |
| 10 | 36 | Register 4 contents. |
| 11 | 40 | Register 5 contents. |
| 12 | 44 | Register 6 contents. |
| 13 | 48 | Register 7 contents. |
| 14 | 52 | Register 8 contents. |
| 15 | 56 | Register 9 contents. |
| 16 | 60 | Register 10 contents. |
| 17 | 64 | Register 11 contents. |
| 18 | 68 | Register 12 contents. |

## CALLING PROGRAM RESPONSIBILITIES AND CONSIDERATIONS

The calling program is responsible for the following:

(1) Loading register 13 with the address of the save area and setting up the second word of the save area.

(2) Loading register 14 with the return address.

(3) Loading register 15 with the entry point address.

(4) Loading registers 0 and 1 with the parameters in an R-type call or loading register 1 with the address of the parameter list in an S-type call.

(5) Saving floating-point registers, if necessary.

(6) Transferring to the entry point of the subroutine.

(7) Restoring floating-point registers, if necessary.

(8) Testing the return code in register 15, if desired.

After the return from a subroutine, the status of the program will be as follows:

(1) In general, the contents of the floating-point registers will be unpredictable unless saved and restored by the calling program.

(2) The contents of general registers 2 through 14 will be restored to their contents at the time the called program was entered.

(3) The program mask will be unchanged.

(4) The contents of general registers 0, 1, and 15 may be changed.

(5) The condition code may be changed.

Note that general registers 0 and 1 and floating-point register 0 may contain results in the case of R-type subroutine calls or a function subprogram. General register 15 will normally contain a return code, indicating whether or not an exceptional condition occurred during processing of the called program.

## CALLED PROGRAM RESPONSIBILITIES AND CONSIDERATIONS

The called program is responsible for the following:

(1) Saving the contents of general registers 2 through 12 and 14 in the save area provided by the calling program. These registers need be saved only if the called program modifies these registers.

(2) Setting up the third word of the save area with the address of the save area which will be provided to subroutines it will call.

(3) Restoring the contents of general registers 2 through 14 before returning to the calling program.

(4) Restoring the program mask if changed.

(5) Loading general registers 0 and 1 or floating-point register 0 with the result in the case of R-type subroutine calls or a function subprogram.

(6) Loading general register 15 with the return code.

(7) Transferring to the return location.

## EXAMPLE CALLING SEQUENCES

This section will describe and give the assembly language statements for the typical machine instructions necessary to implement the calling conventions.

A typical entry point might consist of the following statements:

```
          USING SUBRA,R12          12 will be a base register
SUBRA     STM   R14,R12,12(R13)    save registers
          LR    R12,R15            set up 12 as the base register
          LA    R15,SAVE           save area provided for others
          ST    R15,8(0,R13)       set up forward pointer
          ST    R13,4(0,R15)       set up backward pointer
          LR    R13,R15            set up for any calls we issue
          LR    R11,R1             get parameter pointer into a
*                                  non-volatile register
          .
          .
          .
SAVE      DS    18F                save area we provide for others
```

Inside a subroutine that began with the entry sequence given above, the value of the second parameter in the parameter list could be put into general-purpose register zero with the following sequence:

```
          .
          .
          L     R3,4(0,R11)     pick up second address from par list
          L     R0,0(0,R3)      pick up value of parameter
          .
          .
```

Inside a subroutine that began with the entry sequence given above, another subroutine, SUBRB, could be called using the following sequence. Remember that register 13 already points to the correct save area:

```
            .
            .
            LA    R1,PARLIST       set up parameter list address
            L     R15,=V(SUBRB)    set up entry point address
            BALR  R14,R15          set up return address and branch to
*                                  the subroutine
            B     *+4(R15)       test return code via a transfer table
            B     AOK              return code zero
            B     BAD1             return code four
            B     BAD2             return code eight
            .
            .
AOK         ...                    normal return to here
            .
            .
PARLIST     DC    A(PAR1)          first parameter address
            DC    A(PAR2)          second parameter address
            DC    A(PAR3)          third parameter address
           .
```

Finally, a subroutine that began with the entry sequence given above could return to the program that called it with the following sequence:

```
            LE    FR0,RESULT       floating point result to FPR 0
            L     R13,4(0,R13)     use back pointer to get save area
            LM    R14,R12,12(R13)  restore registers
            SR    R15,R15          zero return code -- no errors
            BR    R14              return to that which called us
            .
            .
```

It should be pointed out that although the above sequences are typical of the instructions used to implement the calling conventions, many variations are possible.

## MACROS FOR CALLING SEQUENCES

There are macro definitions in the MTS macro library *SYSMAC which can be used to help generate calling sequences. The most useful of these macros are ENTER, CALL, and EXIT. For details, see the macro descriptions in MTS Volume 14, 360/370 Assemblers in MTS.

The example given above is repeated below using the REQU, ENTER, CALL, and EXIT macros. The full outline of the assembler program is shown here. This is the way an assembly language subroutine is set up to be accessed from Algol W via a Call statement.

```
SUBRA    CSECT
         REQU  TYPE=DEC
         ENTER R12,SA=SAVE
         LA    R11,R1           protect parameter list address
         .
         .
         L     R3,4(R11)        load address of parameter from list
         L     R0,0(R3)         load parameter using address
         .
         .
         CALL  SUBRB,(PAR1,PAR2,PAR3)
         B     *+4(15)
         B     AOK              return code zero
         B     BAD1             return code four
         B     BAD2             return code eight
         .
         .
AOK      ...                    normal return to here
         .
         .
         LE    FR0,RESULT       load result
         EXIT
         LTORG
SAVE     DS    18F              standard save area
RESULT   DS    E                space for result
         .                      (other storage required)
         .
         END
```

The CALL macro generates its own parameter list, hence the parameter list specified by PARLIST in the original example need not appear in the macro example. In Algol W, this subroutine might be called by a sequence such as the following:

```
integer Jjj, Qqsv;
 .
 .
Qqsv := 1234;
Call("SUBRA", Jjj, Qqsv);
```

After the two load instructions have recovered the second parameter value, general register zero will contain 1234 in binary two's complement representation.

## ALGOL W AND SUBROUTINE CALLING CONVENTIONS

### Internal Calling Conventions

Within Algol W, a set of calling conventions are defined which allow Algol W procedures to call each other. These calling conventions are not the O/S Type I variety. Algol W needs facilities which are outside the scope of these conventions, such as passing name parameters or partial arrays. When the language was designed a set of conventions were agreed upon which are particularly suited to Algol W.

Algol W also allows precompiled procedures to be called via the calling conventions. This entails a small amount of overhead needed to establish the Algol W environment

Fortunately most actions which cannot be performed adequately by Algol W can be achieved by writing a subroutine in assembly language or FORTRAN, and calling it from Algol W via the predeclared procedures Call or Rcall.

### The Call Predeclared Procedure

In Algol W, the Call predeclared procedure implements the full O/S Type I, S-type linkage. Certain additional points are worthy of note:

(1) Parameter lists built by an Algol W Call statement are always of the variable length form, with bit zero set to one on the last parameter address. Since a subroutine must inspect this bit to take action on it, this behavior does not in any way affect the call of a subroutine expecting a fixed length parameter list. It does however allow an Algol W programmer to take advantage of this facility when calling subroutines which do specify that parameter lists may be incomplete. Where MTS system subroutines accept variable length parameter lists this is always clearly specified in MTS Volume 3, System Subroutine Descriptions.

(2) When a subroutine is called with no parameters given, general register zero will contain zero on entry to the subroutine. This is an extension of the variable length convention followed by both FORTRAN and Algol W.

(3) While the O/S Type I calling convention specifies that the program mask should be restored on exit to the pattern on entry, in practice most subroutine authors tend to assume that all four mask bits will be zero on entry. By default, Algol W will have the fixed point (integer) overflow bit set, and reference assignments to the predeclared Exception references may cause other bits to become set.

To avoid unexpected behavior when calling such subroutines, Algol W takes care of its own saving and restoring of the program mask when a subroutine is called. The program mask bits are all zeroed before the call.

Note that this means that integer overflow cannot be detected when calling a FORTRAN subroutine. FORTRAN (sub-)programs never trap this exception, whereas Algol W would report an error by default.

(4) The Algol W system is aware that control has passed to an external subroutine. This knowledge is only used by the error processor if a program interrupt occurs during execution of the called routine. A register dump is produced, and Algol W tries to report the source program coordinate of the Call statement which transferred control.

## The Rcall Predeclared Procedure

Rcall also obeys the O/S Type I calling convention. In its case, of course, no parameter list is set up. The values in general registers zero and one on entry to the subroutine will be those contained in the predeclared integer variables R0 and R1 at the time of the call.

The comments in the previous sub-section on saving and restoring the program mask also apply to Rcall. The program mask bits will always be zero on entry to the subroutine.

## Call and Rcall - Returned Values

As explained in the sections of the main manual describing these routines, function values and the return code can be retrieved on return by inspecting the relevant predeclared variables.

The following correspondence applies between registers on return and the Algol W predeclared variable names:

| Register | Predeclared variable |
|---|---|
| GR0 | R0, R01(0\|4) |
| GR1 | R1, R01(4\|4) |
| GR15 | R_Code |
| FR0 | R_Float, Longrealpart(R_Cmplx) |
| FR2 | Longimagpart(R_Cmplx) |

The instructions which save these registers are in a small routine in Algol W system coding reached by a BAL instruction in the user program code. This is to reduce the amount of code generated for each Call or Rcall statement.

INDEX

Reader's Comment Form

ALGOL W in MTS
Volume 16
September 1980

Errors noted in publication:

Suggestions for improvement:

Your comments will be much appreciated. The completed form may be sent to the Computing Center by Campus Mail or U.S. Mail, or dropped in the Suggestion Box at the Computing Center, NUBS, or BSAD.

Date _____

Name _____

Address _____

_____

_____

Publications
Computing Center
University of Michigan
Ann Arbor, Michigan 48109

Update Request Form

ALGOL W in MTS
Volume 16
September 1980

Updates to this manual will be issued periodically as errors are noted or as changes are made to MTS. If you desire to have these updates mailed to you, please submit this form.

Updates are also available in the memo files at both the Computing Center and NUBS; there you may obtain any updates to this volume that may have been issued before the Computing Center receives your form. Please indicate below if you desire to have the Computing Center mail to you any previously issued updates.

Name ———————————————————————————

Address ———————————————————————————

———————————————————————————

———————————————————————————

Previous updates needed (if applicable):———————

The completed form may be sent to the Computing Center by Campus Mail or U.S. Mail, or dropped in the Suggestion Box at the Computing Center, NUBS, or BSAD. Campus Mail addresses should be given for local users.

Publications
Computing Center
The University of Michigan
Ann Arbor, Michigan 48109

Users associated with other MTS installations (except the University of British Columbia) should return this form to their respective installations. Addresses are given on the reverse side.

Addresses of other MTS installations:

The University of Alberta
Information Coordinator
352 General Services Bldg.
Edmonton, Alberta
Canada T6G 2H1

Information Officer, NUMAC
Computing Laboratory
The University of Newcastle upon Tyne
Newcastle upon Tyne
England NE1 7RU

Rensselaer Polytechnic Institute
Documentation Librarian
130 Amos Eaton Hall
Troy, New York 12181

Simon Fraser University
Computing Centre
User Services Information Group
Burnaby, British Columbia
Canada V5A 1S6

Wayne State University
Computing Services Center
Academic Services Documentation Librarian
5950 Cass Ave.
Detroit, Michigan 48202