UBC PLUS

The Plus Programming Language

by Alan Ballard and Paul Whaley

October 1976

Revised October 16, 1987

Computing Centre

UNIVERSITY OF BRITISH COLUMBIA

6356 Agricultural Road

Vancouver, B.C., Canada V6T 1W5

Note

This manual provides a description of the systems programming language Plus, developed at the University of British Columbia by Alan Ballard and Paul Whaley.

This document was originally prepared for the MTS Systems Workshop at Ann Arbor (October 1976). A major revision including many more details and examples was prepared for June, 1986.

This edition incorporates information on the Motorola 68000 Plus compiler. It corresponds to version 28/13 of the Plus compilers.

The implementations of Plus are not yet complete. This version of the manual has been annotated to indicate language features which are not yet implemented in the compilers.

Plus/370 is fully supported by the UBC Computing Centre for use under MTS. Plus-11 has not been officially released to Computing Centre users, and the external support level has not yet been established. Plus/68000 is currently undergoing initial testing.

Table of Contents

I. The Plus Programming Language

A. Background

Plus is based to a large extent on the Sue system language, which was developed at the University of Toronto, circa 1971, for the specific purpose of implementing an operating system for the IBM System 360 computers.[1] The Sue language was derived (particularly in its data structure facilities) from Pascal. The same is true of Plus, although we have probably moved a little further from the actual syntax of Pascal. The Sue language had a number of awkward constructs and other syntactic rough-spots which we have tried to smooth over. Undoubtedly, some of the changes we've made reflect personal biases, and will not be unanimously viewed as improvements on either Pascal or Sue.

Plus is superficially quite different from Sue or Pascal; however the underlying language semantics are really very similar. It also has much in common with the structure and semantics of the programming language C. If you're familiar with any of these languages, you shouldn't have much trouble adapting to Plus.

The Plus compilers are written entirely in Plus. There are currently three versions, generating code for the IBM System 370-compatible computers, for DEC PDP-11's, and for the Motorola M68000 family of microprocessors. All run on System 370-compatible computers under MTS.[2] The code generated by the 370 compiler is a standard MTS/IBM object module. It does not depend on MTS operating system services, except for a small initialization procedure which must allocate storage. The PDP-11 cross-compiler generates object code for the PDP-11 series in the form of a `*Link11` object module. The code generated does not depend on the system running in the PDP-11, but expects a small run-time environment which is provided by assembler routines. The M68000 cross-compiler generates object code for the Macintosh Programmer's Workshop, or the Macintosh 68000 Development System, or the AMIGA system.[3]

B. Language Goals

This section describes some of the design considerations and language philosophy of Plus. Most of these goals (and in fact, parts of the following description) have been borrowed directly from Sue. The overriding considerations are that the language must contribute to the production of correct, easily maintained, efficient programs. This has a number of implications:

1. Program Structures

The language provides only control structures which encourage reasonable program structure. It must provide an efficient procedure calling/entry sequence, in order to encourage modularity.

2. Problem-Oriented Data Structures

The data structuring facilities of Plus are similar to those of Pascal. They allow de-

[1] IBM is a trademark of International Business Machines Inc.

[2] There is also a version of the 370 compiler which runs using a "fake MTS" interface under IBM's VS-1 or MVS operating systems. The interface was developed by Peter Ludemann of the Block Brothers Data Processing Centre in Vancouver, B.C.

[3] AMIGA is a trademark of Commodore-Amiga Inc., Macintosh is a trademark licensed to Apple Computer Inc.

scription of variables in more problem-oriented terms than most languages. That is, any item is described in terms of the values it will be assigned rather than the storage locations it is to occupy. For example, a numeric variable is defined in terms of the range of numbers it may be assigned, rather than the number of bits it requires. The data structuring facilities tend to be more self-documenting than most languages. They also assist the compiler in making intelligent storage allocation and code generation decisions since the compiler is provided with a description of the essential properties of variables. The information is also available for use in compile-time or run-time checking.

3. Readability and Understandability

This is an important requirement of the language and has influenced its design in many ways. Plus makes no attempt to be terse. In this respect our language differs in a major, philosophical, irreconcilable way from what many people consider a "good" language. Wherever possible, the language uses English keywords rather than strange punctuation marks. Keywords cannot be abbreviated. The language encourages the use of long, self-documenting variable names and symbolic constants. While we haven't yet found a way of preventing the use of Fortran-style remove-all-the-vowels contractions, they are a violation of the intended style.[4]

A number of features commonly found in modern programming languages—such as the ability to have assignments nested within more complex expressions—have been left out of Plus because we believe they lead to programs which are unnecessarily difficult to understand. The small saving in source program size, and possible saving in object code size (we hope our compiler will eventually catch most of the situations where there would be a saving) doesn't seem to justify the inclusion of such features.

To increase the readability of source program listings, the compiler produces only a *paragraphed* listing showing control structures, etc. by means of indentation. The compiler can also produce a paragraphed copy of the source input that is suitable for use as input for subsequent runs of the compiler.

It is particularly important for systems programs that another programmer should be able to pick up a listing and quickly acquire a general idea of how the program works. We believe that the Plus control structures and data structures, together with reasonable identifiers and paragraphed listings, help considerably in achieving this.

4. Parameterization

To reduce maintenance difficulties, it is generally important that a program be written to contain as few occurrences of "magic numbers" as possible. As a general rule such things as sizes of tables etc., should be defined in one place only and referenced wherever else they are required by means of symbolic constants and expressions using the constants. Plus contains the ability to do this kind of parameterization.

5. Compile-Time and Run-Time Checking

The language is expected to assist actively in the detection and isolation of errors, at compile-time if possible. To this end, the language requires all variables to be declared

[4] Within the Plus compiler itself, the average identifier length is about 13.5 characters—longer than the maximum allowed by many languages.

and performs full type checking of all expressions at compile-time. Type checking includes checking the types of parameters to procedures and the results of following pointers.

The compiler will also (optionally) generate extra code to check at run time for certain errors that cannot be detected at compile time. In particular, it can check that values assigned to variables or used as subscripts are within the declared ranges.

Writing the extensive declarations required is often a lot of work. However, our experience with Plus has been that the checking the compiler performs will catch many of the more common programming errors. Also, the extra care that is required in writing a program according to Plus's stringent rules seems to result in programs with amazingly few bugs. We often find that it takes several runs to get a large program to compile successfully, but the resulting object code will work after only a very few attempts at running it.

6. Efficient Code

A systems programming language must necessarily produce efficient code. One consequence of this is that we attempt to avoid including in the language any constructs that are inherently inefficient, or which produce object code larger than you might expect from the source code.

A number of features of the language—including the control structures provided and the nature of the declarations—allow the compiler to obtain information required to produce good code.

The current version of the Plus compiler will generate quite good code on a statement-by-statement basis, but is not clever enough to optimize its register use. Plus allows you to specify that certain variables should be allocated in registers. This means you can assist the compiler in generating good code in critical areas of the program. We hope that future compiler development will provide improvements in the compiler's use of registers, and thus obviate the need for the programmer to specify register variables.

7. Systems Programming Facilities

When you're using a high-level language you shouldn't normally need to worry about machine idiosyncrasies. However there are some situations where—either for reasons of efficiency, or to interface with the hardware or external software—you may need to exercise precise control over the instructions generated and the allocation of storage and registers. There are also situations where, primarily for efficiency reasons, certain language restrictions may be unacceptable. The language provides facilities which can be used, if required, to control size and alignment of variables, force allocation of registers, or emit explicit machine instructions. It also includes the ability to circumvent the usual type-checking rules of the language.

8. Compiler Efficiency

The Plus compiler is reasonably efficient, although this has not been a strong requirement. It's hard to do worse than the 370 assemblers. In fact Plus programs are generally considerably cheaper to compile than equivalent size Assembler programs.

The language provides facilities for separate compilation of parts of a program, in order to keep compilation costs reasonable for large programs.

Some decisions in the compiler implementation have been affected by cost considerations for use under MTS—for example we make extensive use of virtual memory rather than scratch files. Some restrictions of the language result from compilation cost considerations.

II. Tutorial Introduction

This chapter provides a quick introduction to Plus. It will show you the basic elements of the language by presenting and explaining a couple of complete programs.

By the time you've read this chapter, you should be able to write simple Plus programs for yourself. However, this tutorial is not at all complete, and not totally truthful, so if you want to use Plus for real problems, please read the following chapters as well.

Most of what is described in this chapter applies to all the Plus compilers. However, several of the library definitions and procedures described may only exist for Plus/370.

A. A Program to Copy a File

We'll start with a really simple example. Example 1 contains a complete Plus program. The numbers at the left are just for reference in the following explanations; they are not part of the program.

The purpose of this program is to copy one file to another, then write out a count of the number of lines copied. We'll first explain the pieces of this program very briefly, then fill in some details later.

Lines 1 to 3 and line 5 are used to include a number of statements from a library of standard definitions. Library members typically contain declarations for various constants, types, and procedures. Symbols beginning with `%` are compiler variables or compiler procedures. They are used to request special compile-time services.

Line 5 includes a declaration for the procedure `Main` from the library. `Main` is a standard name for the "main program", i.e., the procedure which is going to begin executing when you run the program. Some special magic is required in the declaration of the main program to get things started up properly when you execute the program. The declaration in the library provides the necessary specifications. Later on, when you learn the details of the declaration of `Main`, you can call your main program something else, if you want to.

Lines 7 though 10 are a comment. Anything between `/*` and `*/` is ignored by the compiler. Other comments appear at lines 15–16, line 21, and line 27.

Lines 11 through 30 constitute the definition of the procedure `Main`. Between the heading `definition Main` and the ending `end Main` go declarations of identifiers that are private to `Main`, and the executable statements that are to be performed whenever procedure `Main` is invoked.

Lines 12 and 13 declare three variables, `Count`, `Return_Code`, and `String`, which are to be used in the following statements. `Count` and `Return_Code` are defined to be of type `Integer`, while `String` is defined to be of type `Varying_String`. We'll say more about these specific types in a few minutes. At this point, note that every variable used in a Plus program must be declared, to associate a type with the variable. The variable can only be used in contexts appropriate to its type.

Line 14 is an assignment statement, which sets the value of the variable `Count` to `0`. Note that Plus uses `:=` for assignment.

Lines 17 through 26 constitute a loop, which continues executing indefinitely, until the exit statement at line 22 is executed. That is, execution continues from line 26 by returning to line 17. When the exit is performed, execution continues at line 28, after the end of the loop.

```
 [1] %Include(String_Types, Numeric_Types);
 [2] %Include(Scards_Varying, Spunch_Varying, Sprint_Varying);
 [3] %Include(Integer_To_Varying);
 [4]
 [5] %Include(Main);
 [6]
 [7] /* This is an example program.  It copies an input file
 [8]    to an output file and prints a message saying how
 [9]    many records were copied.
[10] */
[11] definition Main
[12]    variables Count, Return_Code are Integer,
[13]       String is Varying_String;
[14]    Count := 0;
[15]    /* Loop reading and writing records, counting number
[16]       copied. */
[17]    cycle
[18]       Scards_Varying(String, Return_Code);
[19]       if Return_Code ¬= 0
[20]       then
[21]          /* Terminate loop when no more input */
[22]          exit
[23]       end if;
[24]       Spunch_Varying(String);
[25]       Count := Count + 1
[26]    end cycle;
[27]    /* Build and print a message. */
[28]    String := " Copied " || Integer_To_Varying(Count, 0) || " records.";
[29]    Sprint_Varying(String)
[30] end Main
```

Example 1—File Copy Program

Line 18 invokes a procedure[1] `Scards_Varying`, whose definition was included from the library by line 2. `Scards_Varying` reads a line from MTS I/O unit `Scards`, and assigns the value read to the first parameter, which must be a suitable character variable. It assigns the return code of the operation to the second parameter. Note that the procedure call just consists of the name of the procedure followed by a list of its parameters. There is no "call" keyword.

Lines 19 through 23 constitute an if statement. Note that the if statement is terminated with an `end if`. (Just `end` is allowed also.) Between the keyword `then` and the `end if`, there could be an arbitrary list of statements, although in this case there is just the exit statement. In general, there might also be an else-part before the `end if`. This particular if statement tests the variable `Return_Code`, which `Scards_Varying` sets to the return code delivered by the `Scards` call. If it is nonzero (indicating end-of-file or an error), then the loop is terminated.

[1] To be strictly accurate, `Scards_Varying` is not a procedure but a *macro*. This distinction is unimportant at this point, and in fact we won't discuss macros in this tutorial.

Line 22, the exit statement, terminates execution of the loop. We could actually have specified the condition as part of the exit statement, replacing all of lines 19 to 23 with

```
exit when Return_Code ¬= 0;
```

Line 24 writes out the line just read to I/O unit `Spunch`.

Line 25 adds one to the counter of lines read. Actually, this statement isn't very good Plus. Instead, it should be written as

```
Count +:= 1
```

which means the same as

```
Count := Count + 1
```

but may be more efficient (besides being less typing).

Line 28 builds a message specifying the number of records copied. The operator `||` is used to concatenate character strings. `Integer_To_Varying` is a library function (whose declaration was included at line 3) which converts an integer to a character string. The parameters of `Integer_To_Varying` specify the number to be converted, and a field width to use. 0 for the field width means to format it in the minimum number of characters. Note that functions in Plus can return arbitrary objects—in this case, a variable length character string is returned.

Line 29 then prints out this message on I/O unit `Sprint`, using another library routine.

After executing the last statement, the procedure automatically returns to its caller—in this case terminating execution and returning to the operating system.

B. Compiling and Running the Program

If this program is in the file `Plusex1.s`, it can be compiled by issuing the MTS command

```
Run *Plus Scards=Plusex1.s
```

This will produce a listing on I/O unit `Sprint` (which normally defaults to your terminal or printed output), and an object module in the file `-Load`. You can specify I/O unit `Spunch` on the `Run` command to put the object module somewhere else.

The program (from file `-Load`) can be executed by issuing a command like

```
Run -Load Scards=Infile Spunch=Outfile
```

which copies file `Infile` to `Outfile`.

C. Program Format

Plus programs are free-format, with line boundaries being ignored. You may format the text in your source file any way you like, breaking lines wherever is convenient, except that you cannot break in the middle of a single "token". That means you can't split an identifier, a keyword, a string constant, etc., across two lines. (A comment is treated as a sequence of tokens, so it can be continued across any number of lines.)

Statements are separated with semicolons. Thus, for example, lines 1, 14, 18, contain one statement each. Line 17, on the other hand, is not a statement. It is just a part of the compound statement which goes from line 17 to line 26. We'll say more later about just when you need a semicolon and when you don't. In general, Plus is fairly forgiving if you put in some you don't really need. All the examples in this manual will include exactly the semicolons that are required, with no extra ones.

The listing produced by the Plus compiler will always be formatted to indicate the structure of your program, indenting the insides of loops, and so on. All the examples in this manual are formatted the way they would be by the compiler. You can also ask the compiler to produce a copy of your program that is formatted in the same way, by assigning unit 1 on the `Run` command. This can be useful to produce a cleaned-up source file after you've edited a program extensively.

Plus uses quite a lot of keywords to define the various kinds of statements, operations and types, and to make them as readable as possible. These keywords are reserved—you can't use them as names for things you define in your program. A complete list of Plus's reserved words appears in Appendix C. In the example program, and throughout this manual, the reserved words appear in all lowercase letters, while symbols which are not reserved will have the first letter of each segment capitalized. So `if`, `cycle`, `program` are reserved, but `Integer`, `Return_Code`, `Numeric_Types` are not. This same lowercase/capitalization convention is used by the Plus compiler in any messages it produces. By using the compiler option `%Lower_Case`, you can ask the compiler to format identifiers and keywords in the listing and reformatted copy this way too.

This is *only* a convention, however. The compiler ignores upper/lower case distinctions in interpreting its input. That is, `IF`, `iF`, `If`, and `if` are all interpreted as the reserved word `if`, and `Count`, `count`, `COUNT`, etc., are all the same variable.

D. Declarations in Plus

In Plus, as in most languages, you will use identifiers for several different purposes—the names of variables (of various types), procedures, fields of records, and so on.

Most of these will be names which are invented by you and are specific to your program. You must define *all* such identifiers in an appropriate declaration statement.

Some identifiers may be defined by including their declarations from a **source library**. There are a few others that are predefined by the compiler, but not reserved—you can use the same symbol for your own purposes (although you probably shouldn't).

The various kinds of declarations can appear in any order, and can be intermixed with executable statements. However, every identifier has to be declared before the first time it is used in other contexts.

E. The Source Library

To reduce repetitive coding, and to help maintain consistent definitions, the Plus compilers provide a source library facility. The `%Include` procedure is used to input Plus source code from such a library. In effect, the `%Include` statement is replaced by the contents of the library members whose names are given as parameters.

The file `*Plus.Sourcelib` is a standard library that is searched by default. It contains definitions of some useful constants and types, the declarations of a number of library routines (such as `Integer_To_Varying`), and declarations for most of the MTS system subroutines. Each library member in turn includes any other declarations on which it depends.

The definitions in `*Plus.Sourcelib` are documented by a separate writeup (**UBC PLUS LIBRARY**). This tutorial will just describe a few of the more basic members.

If you're writing a large Plus program, you will probably want to divide it up into a number of files, which you can then separately compile and modify. You should put common declarations in a private library, from which they can be can be `%Included` by each component, just like declarations from the standard library. The libraries to be searched by the compiler are specified on I/O unit `0` of the `Run` command. For example, if you want the private library `Mylib` searched as well as the default library, you would enter

```
Run *Plus ... 0=Mylib+*Plus.Sourcelib
```

Chapter VII (page 135) describes the format of a Plus library, as well as the program `Plus:Libgen` which can be used to generate a library.

F. The Runtime Library

There is a library of procedures that are often used by Plus programs (such as `Integer_To_Varying`). These library procedures are in the MTS resident system. In order for the system loader to find them when it loads your program, some special loader records must be present in your object file (e.g. `-Load`). The declaration for `Main` that was included by line 5 also causes these records to be added to the object file, so this simple program will run as is. However, when you deal with more complex programs that have been compiled as a number of separate pieces, you may have to make sure the records needed are present and in the right place (generally at the end of the file). Details of this are in Chapter IV (see page 127).

G. Types and Declarations

The program in Example 1 declares variables of two types, `Integer` and `Varying_String`. As the capitalization indicates, these are not reserved words. In fact, they are not built-in types at all. Rather, the definitions of these two types also come from the library, as a result of the `%Include` at line 1.

It might seem strange that these apparently basic types are not predefined as part of the language. However, in Plus they are not really particularly basic, as we will indicate in the following sections.

1. Numeric Types

The library member `Numeric_Types` that is included at line 1 contains a number of declarations that relate to integer numeric processing.

The following declarations are among those in the library member:

```
constant Maximum_Integer is 2147483647,
   Minimum_Integer is -Maximum_Integer - 1;
type Integer is (Minimum_Integer to Maximum_Integer)
```

The first statement (from the keyword `constant` to the semicolon at the end of the second line) is an example of a Plus **constant declaration**. Such a statement just associates one or more identifiers with constants. Thereafter, the identifier may be used instead of repeating the constant. Thus the identifier `Maximum_Integer` refers to the constant 2147483647 (which is, indeed, the maximum integer available on the 370 computers).

Having defined `Maximum_Integer`, you can use it in any context where the constant would be allowed, with equivalent effect.

The value associated with the identifier in a constant declaration may be an expression, as long as all the elements of the expression are themselves constants. That is, the compiler must be able to determine the value to be associated with the identifier. The identifier `Minimum_Integer` is associated with the value $-2147483647 - 1 = -2147483648$.[2]

The second statement (beginning with the keyword `type`) is an example of a **type declaration**. It associates the identifier `Integer` with the `type` represented by the phrase following `is`. The identifier `Integer` can be used thereafter instead of repeating the phrase `(Minimum_Integer to Maximum_Integer)`.

This example illustrates the basic mechanism you use in Plus to define a numeric type. You specify a range of values that are to be allowed for variables of the type. So the declaration

```
variable Count is Integer
```

is equivalent to

```
variable Count is (Minimum_Integer to Maximum_Integer)
```

and means that variable `Count` may be legitimately assigned any number in the indicated range.

The compiler will (optionally) generate run-time tests to ensure that assignments to numeric variables obey the range limitation indicated in the declaration. This run-time checking is often very effective in detecting bugs in programs at an early stage. You should be as precise as possible in defining the range of variables, since this gives the compiler the most opportunity to be helpful. (Moreover, the compiler will take advantage of range information in some cases to improve the code generated.)

It should really be quite rare for you to need to use the type `Integer`. In fact, even for the example program, it would be preferable to define `Count` as `(0 to Maximum_Integer)`, since `Count` should never acquire a negative value.

2. String Types

The library member `String_Types` includes the following declarations:

```
constant Standard_String_Length is 255;
type Fixed_String is character(Standard_String_Length),
   Varying_String is character(0 to Standard_String_Length)
```

The type declaration provides examples of two forms of character string types implemented by Plus.

An example of a fixed-length string type is provided by

```
character(Standard_String_Length)
```

Variables of this type always contain exactly 255 characters (the value of the constant `Standard_String_Length`). Assigning a longer value will generate an error message,

[2] For obscure reasons, you can't write this constant directly into a program, but you can get at it indirectly as in this example.

while a shorter value will leave the last characters unchanged. (Short strings will not be padded to the given length.)

A varying-length string type is illustrated by

```
character(0 to Standard_String_Length)
```

Variables of this type may be assigned character string values containing anything from 0 to 255 characters. The variable will keep track of the length of the value last assigned.

The length 255 has been somewhat arbitrarily picked as the size of these string types in the source library definition. This is generally adequate for most programs for building messages, etc., without wasting too much memory.

Plus allows character types (either fixed or varying length) of any length—they are *not* restricted to the length 255. Note, however, that when a variable of a varying-length character type is declared, enough memory is allocated for the maximum length specified (plus a length field). So you should avoid defining overly-large character variables when possible.

H. Plus I/O

The Plus language doesn't include any input/output statements. Instead, it is assumed that the system I/O subroutines such as `Read`, `Write`, `Scards`, etc. will be used.

To facilitate use of these routines, there are several definitions in `*Plus.Sourcelib`. These include declarations necessary to use the subroutines directly, as well as some interfacing declarations that simplify things for common situations.

The routines `Scards_Varying`, `Spunch_Varying`, `Sprint_Varying` and similar ones for other I/O units, provide a simple interface for reading and writing standard `Varying_String` variables, as described above. Other routines provide for I/O to or from arbitrary variables in a Plus program.

The Plus library includes a rather powerful procedure `Message` for producing formatted output (generally, "readable" messages). The `Message` routine can perform a variety of conversions and substitute the results into a string which it then prints. The next example program illustrates some simple uses of `Message`, but for all the details, see the separate documentation for the routine.

The conversion routines in the Plus runtime library may also be useful in building formatted lines to be written.

The Plus library also includes a few simple routines for performing "input conversions"– e.g., `String_To_Integer` will convert a character string to an integer. However, most Plus programmers prefer to use `Clparser`, the MTS "Command Language Parser" subroutine package, to perform input processing. `Clparser` is described in the writeup **UBC CLPARSER**.

I. Table Search Example

The following pages contain a much more substantial example, containing most of the elements of a typical Plus program. The program is a demonstration of linear and binary searching algorithms. It first reads a table of symbols, terminated with `/end`. Then it reads symbols to look-up, searches the table using each of linear and binary search techniques, and prints out the symbol position and number of accesses required to find it.

```
%Title := "Plus Example Program - Linear and Binary Searching";
/*frame,centre
                    Linear and Binary Searching
*//*
   This example program demonstrates linear and binary
   searching.  It first reads a table of "symbols",
   then reads "test cases" and looks each up in the table
   by both search techniques.  It prints the position
   of the search item, and how many "probes" it took to
  find it.
*/
%Include(Numeric_Types, String_Types);
%Include(Scards_Varying);
%Include(Message_Initialize, Message, Message_Terminate);

global Search_Example
   /* Define limits on symbol length and number of
      symbols. */
   constant Max_Sym_Length is 10;
   constant Max_Number_Symbols is 600;

   /* Define types for symbols, table, etc. */
   type Symbol is character(0 to Max_Sym_Length);
   type Array_Index is (0 to Max_Number_Symbols),
      Symbol_Array is array (1 to Max_Number_Symbols) of Symbol;

   /* Declare the table that the symbols will be entered
      into. */
   variable Table is Symbol_Array,
      Table_Size is Array_Index;

   /* Declare control block for the message routine. */
   variable Msg is pointer to Stream_Type

end global Search_Example;

%Include(Main);

procedure Getsym is
      procedure
      result Sym is Symbol
      end;

```

Example 2 (part 1 of 5)—Table Search Program

```
procedure Print_Result is
      procedure
      parameter Sym is Symbol,
         Method is character(1 to 10),
         Pos is Array_Index,
         Accesses is Integer
      end;

procedures Linearsearch, Binarysearch are
      procedure
      parameter Element is Symbol,
      reference parameter Accesses is Integer
      result Position is Array_Index
      end;

%Eject();
definition Main
   /* Main program reads in symbol table, terminated by /end, then reads
      test cases and finds them by both linear and binary search.
   */
   variable Return_Code is Integer;

   /* Initialize the Message routines. */
   Msg := Message_Initialize();

   /* Display title and prompts... */
   Message(Msg, "*** Demonstrate Linear and Binary Search ***</>");
   Message(Msg, "Enter symbol elements (in alphabetical order)</>");
   Table_Size := 0;

   /* Read in the test table. */
   cycle
      variable Elem is Symbol;
      Elem := Getsym();
      exit when Length(Elem) = 0 or Elem = "/end";
      if Table_Size >= Max_Number_Symbols
      then
         Message(Msg, "Error - too many symbols.</>");
         exit
      end if;
      Table_Size +:= 1;
      Table(Table_Size) := Elem
   end cycle;

   Message(Msg, "<hi> data items read.</>", Table_Size);

```

Example 2 (part 2 of 5)—Table Search Program

```
   /* Read in the test cases and look up each. */
   cycle
      variable Test_Elem is Symbol,
         Pos is Array_Index,
         Accesses is Integer;
      Message(Msg, "Enter data:</>");
      Test_Elem := Getsym();
      exit when Test_Elem = "";

      /* Look up using linearsearch and output result. */
      Pos := Linearsearch(Test_Elem, Accesses);
      Print_Result(Test_Elem, "linear", Pos, Accesses);

      /* Look up using binarysearch and output result. */
      Pos := Binarysearch(Test_Elem, Accesses);
      Print_Result(Test_Elem, "binary", Pos, Accesses)
   end cycle;

   Message_Terminate(Msg)
end Main;
%Eject();
definition Getsym
   /* This procedure reads a symbol, and checks for
      invalid strings. It returns null string at eof. */

   variable Str is Varying_String,
      Return_Code is Integer;

   Scards_Varying(Str, Return_Code);

   if Return_Code ¬= 0
   then
      Sym := ""
   elseif Length(Str) > Max_Sym_Length
   then
      Message(Msg, "Error - symbol too long</>");
      Sym := Substring(Str, 0, Max_Sym_Length)
   else
      Sym := Str
   end if
end Getsym;

```

Example 2 (part 3 of 5)—Table Search Program

```
definition Print_Result
   /* Prints out message saying where symbol was found
      and how many accesses it took. Parameters
      are the "search method", the symbol, the
      position and the number of accesses.
   */

   Message(Msg, "<v> search: <v> ", Method, Sym);
   if Pos = 0
   then
      Message(Msg, "not found")
   else
      Message(Msg, "found at <hi>", Pos)
   end if;
   Message(Msg,  " in <i> accesses.</>", Accesses)
end Print_Result;

%Eject();
definition Linearsearch
   /* Search linearly for Element in Table.

      Table and Table_Size are global.  The symbol
      to search for is passed as the parameter Element.

      Returns the position as function result, or 0 if
      the symbol is not found.  Sets reference
      parameter Accesses to the number of probes
      required.
   */
   variable Pos is Array_Index;

   Accesses := 0;
   do Pos := 1 to Table_Size
      Accesses +:= 1;
      return when Table(Pos) = Element with Pos
   end;
   return with 0
end Linearsearch;
```

Example 2 (part 4 of 5)—Table Search Program

```
definition Binarysearch
   /* Search for Element in Table using a binary search.

      Table and Table_Size are global.  The symbol
      to search for is passed as the parameter Element.

      Returns the position as function result, or 0 if
      the symbol is not found.  Sets reference
      parameter Accesses to the number of probes
      required.
   */
   variables Low, High, Pos are Array_Index;
   /* Low and High delimit the range of the table that
      must contain the element, if it is present.  */
   Low := 1;
   High := Table_Size;
   Accesses := 0;

   cycle
      exit when Low > High;
      /* Compute next plce to check (midpoint
         between low and high). */
      Pos := (Low + High) / 2;
      Accesses +:= 1;
      return when Table(Pos) = Element with Pos;
      if Element < Table(Pos)
      then
         /* If element is in table, it must be
            between Low and Pos - 1 */
         High := Pos - 1
      else
         /* If element is in table, it must be
            between Pos + 1 and High. */
         Low := Pos + 1
      end if
   end cycle;
   return with 0

end Binarysearch
```

Example 2 (part 5 of 5)—Table Search Program

The general organization of this program is characteristic of many Plus programs.

It begins with specification of a title to appear in the listing (line 1), and introductory comments (lines 2 to 11). The words `frame,centre`[3] immediately after the opening comment `/*` cause the compiler to draw a frame of asterisks around the comment and to centre each line of the comment in the listing.

Lines 12 through 14 include library definitions, as in the previous example. This example uses the library message formatting routines, whose declarations are included at line 14.

Next come a number of global declarations. Global declarations define identifiers (types, constants and variables) that are to be available to all procedures of the program. They are normally grouped in one or more **global blocks**. Each global block has a name, which is used in associating the same global block across separate compilations of pieces of a program. In this case, there is only one global block, lines 16 to 35.

A program may have any number of global blocks. Generally, you should group related definitions together as one global block.

Following the global blocks are a number of global declarations for the procedures making up the program (lines 37 to 57). You must have a declaration for every procedure that is either defined in, or called from the program. As in Example 1, the declaration for `Main` is included from a library.

The remainder of the program consists of the definitions of the procedures, each beginning with `definition ...` and ending with `end` as in the previous example.

The order in which these pieces occur is typical, and is generally convenient. However, you aren't required to put the pieces in any particular order. It is quite permissible to have some global definitions followed by some procedures, followed by more global definitions, and so on. The only requirement is that you must declare each identifier before its first use.

The remaining sections of this chapter will describe the elements of Plus in a more orderly way, using examples from this program.

J. Program Structure

1. Separate Compilation

A complete Plus program consists of one or more separately compilable pieces. You'll probably want to keep such pieces in separate files. Each piece contains a sequence of declarations, globals blocks and procedure definitions. Procedure definitions may not be nested inside other definitions. (Global blocks *can* be nested inside procedures or other global blocks, although there is rarely any reason to do so.)

You can divide a program up into pieces however you like, except that each piece must be self-contained to the extent of including definitions of all the identifiers it references. Of course, you will usually choose pieces consisting of groups of related procedures and the global definitions they use.

When a global block contains definitions that are required by more than one of the separately-compiled pieces, you must repeat the entire global block with each piece. It's

[3] Americans may substitute `center`.

best to put the definition of the global block in a private source library and use `%Include` to include it in each piece.

If you change any of the declaration in a global block, generally you must recompile *all* components that reference that global block.

2. Global Blocks

A global block contains a sequence of declarative statements, separated by semicolons. Each global block has a name (`Search_Example` in line 16 of Example 2), which becomes an external symbol of the program and is used in associating the definitions from independently-compiled pieces.

A global block is much like a Fortran named-common block, except that it may contain declarations of types, constants and procedures as well as variables. Moreover, Plus globals are usually implemented in a way that is fully reentrant (which Fortran common isn't).

3. Procedure Definitions

A procedure in Plus consists of two parts, a **procedure declaration** which specifies the type of the procedure, and a **procedure definition** which contains declarations and statements to be executed when the procedure is called. Thus, for example, the procedure `Linearsearch` is declared at lines 52–57, and the definition is given at lines 150–169. Note more than one procedure can be declared in the same declaration, when they have identical types.

The **type** of the procedure tells what identifiers are to be used by the definition to refer to its parameters and return value. It also specifies the types of the parameters and return value.

A procedure with a result is a function and is used as an element of expressions. A procedure with no result is a subroutine and is called as a separate statement. `Getsym` (declared at line 39, called at lines 77 and 96) is an example of a function with no parameters. `Linearsearch` and `Binarysearch` are each functions with parameters. `Print_Result` (declared at line 44, called at lines 101 and 105) is an example of a subroutine.

The definition of a procedure must be preceded by its declaration. Any call of a procedure must be preceded by a declaration of the procedure called. The easiest way to ensure the correct ordering is to simply place all procedure declarations before any definitions, as in the examples.

When writing programs that are to be compiled in pieces, you may find it helpful to always place the declarations of procedures in a source library. Then the file containing the procedure definition, and any files containing calls to it, can all include the same declaration.

K. Declarations

There are four important declarative statements in Plus. These are constant, type, variable and procedure declarations. All of these have been illustrated already. In this section, we'll fill in a few more details.

The four declarations have a somewhat similar overall syntax. The basic form of each is illustrated by

```
constant Max_Sym_Length is 10;

type Symbol is character(0 to Max_Sym_Length);

variable Msg is pointer to Stream_Type;

procedure Getsym is
      procedure
      result Sym is Symbol
      end
```

When there are a number of variables to be defined with the same type, you can put a list of identifiers in place of the single identifier in these examples. Thus, for example,

```
variables Low, High, Pos are Array_Index
```

at line 181 is a shorthand for

```
variable Low is Array_Index;
variable High is Array_Index;
variable Pos is Array_Index
```

(The keywords `is` and `are` are equivalent, as are `variable` and `variables`. You can use whichever is grammatically appropriate.)

You can also run a series of variable declarations together. So, for example,

```
variable Table is Symbol_Array,
   Table_Size is Array_Index
```

at lines 29–30, is equivalent to

```
variable Table is Symbol_Array;
variable Table_Size is Array_Index
```

Similar short-cuts are allowed for each of the other kinds of declarations.

1. Scope of Declarations

Identifiers that are declared in a global block (or outside of any procedure definition) are called **global**. Such identifiers may be referenced from any subsequent statement in the program. In the example program, all of the identifiers declared in lines 1–58 are global. Thus, for example, the two search procedures can reference the variable `Table` without it having to be passed to them as a parameter.

Identifiers (type, constant, variable, or procedure) that are declared inside a procedure definition are **local** to that procedure. The declarations are not "known" outside of the procedure. So the declaration of the variable `Return_Code` at line 64 in procedure `Main` can't be referenced outside of `Main`. The procedure `Getsym` has its own variable `Return_Code` declared at line 116. These are two totally different variables, even though they happen to have the same identifier. The references inside `Main` use the variable declared in `Main`, while those inside `Getsym` use the one declared in `Getsym`. If you tried to reference `Return_Code` in `Print_Result` or one of the other procedures, you'd get a complaint from the compiler about it being undeclared.

In fact, the scope of a declaration may be restricted further still. Like Algol and most of its descendants, Plus provides for **scope blocks** which delimit groups of statements within which a given declaration is known. Unlike most such languages, you don't have to use lots of `begin ... end` groups to introduce local declarations. In Plus, the statement list inside each "bracketed" control structure such as the `cycle...end` loop or either part of the `if...then...else...end` statement, is a separate scope block. Declarations that occur inside such a statement list are in effect only for the remainder of the statement list. This makes it easy to introduce a temporary variable at the point where you need it.

For example, somewhere inside a sorting program, you might have a statement of the form

```
if Table(I) > Table(J)
then
   /* Interchange I'th and J'th elements. */
   variable Temp is Symbol_Table_Element;
   Temp := Table(I);
   Table(I) := Table(J);
   Table(J) := Temp
end if
```

The variable `Temp` is declared only for the remainder of the statement list in the then-part. It is undefined outside of the if statement, or in the else-part of the statement (had there been one).

There might already be a definition of `Temp` in effect at the beginning of the if statement, if it had been declared either globally or earlier in the statement list containing the if statement. The definition inside the if statement is still allowed, and temporarily overrides the outer definition. At the end of the then-part, the previous definition comes back into effect.

For example, in a sequence of statements such as

```
[1] variable Temp is Integer;
    ...
[2] Temp := 1;
[3] if ...
[4] then
[5]    variable Temp is Symbol_Table_Element;
[6]    Temp := Table(I);
       ...
[7] end if;
[8] X := Temp
```

the statements at lines 2 and 8 both refer to the variable declared at line 1; its value is not affected by the assignment statement at line 6. The assignment at line 6 refers to the variable declared at line 5. Its value is independent of the variable declared at line 1.

2. Constant Declarations

The constant declaration just lets you associate an identifier with a constant value. From then on, you can use the identifier instead of writing out the constant each time.

This helps to "parameterize" the program so that it is easier to change in the future. Example 2 assumes that the symbols it has to deal with will be no longer than 10, and that there will be no more than 600 of them. If either of these assumptions turned out to be inadequate, you would just have to change the declaration at line 19 or 20 and recompile the program, without having to search through the program looking for all the places that depend on these limits.

A second advantage of using constant declarations is that they often make it easier to understand the program, since the purpose of a well-chosen mnemonic identifier may be much clearer than an anonymous constant.

3. Type Declarations

Similarly, a type declaration lets you associate an identifier with a type description. Once again, this is useful for parameterizing your program, and may make it easier to read.

A type declaration is in some ways like an assembler dsect (especially when the type is a record). It defines a template describing an area of storage, but does not allocate storage. The variable declaration is used to allocate storage for an item of a given type.

Type declarations may be *required* in some situations to associate an identifier with a type description which you need to use in more than one place. Plus doesn't attempt to determine if two complex type descriptions are "equivalent". For example, if you had two record variables (we'll explain records soon) declared as

```
variable Element1 is
      record
         Symbol is Symbol_Type,
         Reference_Count is Integer
      end;

variable Element2 is
      record
         Symbol is Symbol_Type,
         Reference_Count is Integer
      end
```

the types of `Element1` and `Element2` will be considered incompatible, so you wouldn't be able to assign one to the other or otherwise intermix them. You must use the same type definition for each. The usual way to do this is to define the type with a type declaration and then use its name:

```
type Symbol_Table_Element is
      record
         Symbol is Symbol_Type,
         Reference_Count is Integer
      end;
variable Element1 is Symbol_Table_Element;
```

```
    variable Element2 is Symbol_Table_Element
```

(If there is no other requirement for the type `Symbol_Table_Element`, you could alternatively declare both in one variable declaration:

```
    variables Element1, Element2 are
          record
             Symbol is Symbol_Type,
             Reference_Count is Integer
          end
```

But generally, it is better to use the type declaration.)

The requirement that a type description appear in only one place helps to minimize the danger of introducing bugs when the definitions are changed. (It also eliminates a potentially very expensive compile-time action in determining if two definitions are "equivalent".)

4. Variable Declarations

Variable declarations are used to allocate memory for a variable or variables of a specified type. A global variable (one declared in a global block) is allocated once, at the time the program begins execution. A local variable (one declared inside a procedure) is allocated *each* time the scope block containing it begins execution, and is released at the end of the scope block. Thus, for example, a variable declared inside a loop is released at the end of the statement list forming the body of the loop—you may *not* assume it will keep its value from one iteration to the next.

We should perhaps emphasize that there is *no* run-time overhead to allocating and releasing variables inside a scope block—all the storage allocation calculations are done at compile-time. Local variables are allocated using a run-time "stack" mechanism. The only actual allocation overhead occurs at entry to the procedure, at which point the stack is adjusted to allow for all the local variables declared within the procedure and temporary storage required by the generated code. Hence you may freely declare local "temporary" variables at the point where they are required.

5. Procedure Declarations

A procedure declaration is used to specify the type of a procedure. The declaration at lines 52–57 says that both `Linearsearch` and `Binarysearch` are procedures with type

```
    procedure
    parameter Element is Symbol,
    reference parameter Accesses is Integer
    result Position is Array_Index
    end
```

The type of a procedure *must* be a procedure type, as you might expect. (You'll learn some of the details of procedure types in Section L–5, page 27.) Most often, the type will be specified directly in the procedure declaration, as in this example. But as with all other types, it is quite permissible to define the type in a type declaration and use its name. So lines 52–57 could be replaced with

```
type Table_Search_Procedure is
      procedure
      parameter Element is Symbol,
      reference parameter Accesses is Integer
      result Position is Array_Index
      end;
   ...
 procedures Linearsearch, Binarysearch are Table_Search_Procedure
```

A procedure with no parameters and no result may be declared as just

```
procedure Proc
```

which is equivalent to the declaration

```
procedure Proc is
      procedure
      end
```

L. Type Descriptions

A **type description** is a program fragment that you use to define a data type. Plus provides some primitive data types, and some methods of building new types out of simpler ones. In Section G we described a couple of the basic data types of Plus, numeric types and string types. In this section we'll tell you about some of the other types, and how they are used in a program. This is just an outline, however. For all the details, see Chapter III.

Type descriptions appear in several contexts in the language. The most important contexts are variable and type declarations and in the description of more complex types.

1. Basic Types

Plus's numeric and string types were described in Section G. Other basic types in Plus are

a. `bit(`*n*`)`, where *n* is an integer constant. This just describes the specified number of bits of memory. For example:

```
variable X is bit(32)
```

specifies that the variable `X` is to be allocated as 32 bits. Depending on context, bit types may behave like various other types (integers, strings, and others). They are very machine-dependent, so you should generally avoid using them unless you really need to.

b. `real(`*n*`)`, where *n* is an integer constant. This specifies a "floating-point" data item. The number *n* indicates how many digits of precision are required.

You can't do much with floating-point in Plus at the moment, but you can define variables.

c. **identifier-lists**. An identifier-list type allows you to create new basic types by enumerating a list of identifiers which are to be the elements of the type. (Pascal calls this type an "enumerated type".) For example:

```
(Printer, Reader, Punch, Tape_Drive, Disk_Drive, Terminal)
```

If this description appears in a program, it defines a new type. The elements of the identifier list are automatically declared to be symbolic constants of the given type (and must therefore not be previously declared in the same scope). The compiler is free to choose values to represent each of the constants.

Most often, such types appear in a type declaration, so that you have a name with which to refer to them later:

```
type Device_Type is (Printer, Reader, Punch, Tape_Drive,
   Disk_Drive, Terminal)
```

After this declaration you might define a variable, as

```
variable Device is Device_Type
```

assign the variable a value:

```
Device := Disk_Drive
```

test it in if statements:

```
if Device = Reader
then
   ...
end if
```

and so forth.

Identifier-list types are very useful for variables whose values have no intrinsic numeric meaning. Use of these types is a significant aid to writing self-documenting code. These types do the kind of thing a good programmer might do in assembler with equates (except that the compiler rather than the programmer does all the bookkeeping to decide which value to use for each item).

2. Record Types

In this example program, the symbol table to be searched contains just the actual symbols (identifiers). If you were writing a "real" program using a symbol table, however, you would almost certainly need to associate some information with each element of the table. Such a collection of associated information is implemented in Plus, as in many other languages, by using **record types**.

Suppose each entry of the symbol table is to contain the actual symbol and an associated integer which just counts how many times it has been referenced. The elements of the symbol table could then be represented in Plus by the type description

```
record
   Symbol is Symbol_Type,
   Reference_Count is Integer
end
```

This might be used in a type declaration as

```
type Symbol_Table_Element is
      record
         Symbol is Symbol_Type,
         Reference_Count is Integer
      end
```

and the definition of the actual symbol table type (line 25) might be replaced by

```
type Symbol_Array is array(1 to Max_Number_Symbols) of Symbol_Table_Element
```

The individual items in the record type are known as **fields**. Of course, a record type can have any number of fields, and each field can be of any type, including another record type.

The individual fields of a record are referenced using a dot (“`.`”). Continuing the example, if you declare

```
variable Sym is Symbol_Table_Element
```

you can set the fields of the variable as

```
Sym.Symbol := Elem;
Sym.Reference_Count := 0
```

and increment the reference count as

```
Sym.Reference_Count +:= 1
```

and so forth.

3. Array Types

To describe an array in Plus, you specify the range of the allowed subscripts and the type of data item making up the array. So the type description contained in line 25 specifies an array of 600 elements (1 through `Max_Number_Symbols`). Each element of the array in this case is a `Symbol`. Note the lower-bound of an array doesn't have to be one; any range is allowed.

Individual elements of an array are accessed in a program using parentheses around a subscript expression. So

```
Table(Table_Size)
```

in line 85 of the example accesses the element at position `Table_Size` in the array.

An array can be composed of any type of data item. As described in the previous section, a realistic symbol table application might use an array, for which each element is a record. You can create multi-dimensional arrays by using arrays of arrays. If you need to work with 50 by 100 matrices of integers, you might use a type of the form

```
array (1 to 50) of array (1 to 100) of Integer
```

This defines an array of 50 elements (`array (1 to 50) of ...`) each element of which is an array of 100 integers.

If `Matrix` is a variable of such a type, then

```
Matrix(I)
```

refers to the I'th row of the matrix, and

```
Matrix(I)(J)
```

refers to the J'th element of the I'th row. This can also be expressed as

```
Matrix(I, J)
```

Array subscripts can be combined with record field selection in an expression. If the array in the example were replaced with an array of records you might replace line 85 with

```
Table(Table_Size).Symbol := Elem
```

which sets the field `Symbol` of the record at position `Table_Size` in the array.

4. Pointer Types

A **pointer** in Plus is the machine address of a data item. A common use of pointers to build linked-lists and other complex data structures in which each data item "points to" the next item of the list.

For example, if you are building a symbol table, but don't want to have an *a priori* limit on the number of elements in the table, you might choose to use a linked list for the table. (It would, however, be difficult to implement a binary search for such a table. Other techniques for fast searching would be appropriate.) To do so, you'd define each element of the table as a record, something like:

```
type Symbol_Table_Element is
      record
         Next_Symbol is pointer to Symbol_Table_Element,
         Symbol is Symbol_Type,
         Reference_Count is Integer
      end
```

The type description `pointer to Symbol_Table_Element` within the record is a **pointer type**. A pointer type always specifies what type of object it points to, so that when you use the pointer in an expression, Plus knows what type of object it is dealing with.

If you used pointers in this way, instead of defining an array in the global block, you would just define a variable to point to the first element:

```
variable Symbol_Table is pointer to Symbol_Table_Element
```

(You might also want a second variable to keep track of the end of the list.) The list can be initialized to indicate it contains no elements by using the special pointer value `Null`:

```
Symbol_Table := Null
```

To add another element to the list, you would call some kind of a memory allocation procedure. For examples, MTS's `Getspace` routine could be called directly:

```
variable New_Elem is pointer to Symbol_Table_Element;
   ...
New_Elem := Getspace(0, Byte_Size(Symbol_Table_Element))
```

Here, a built-in procedure `Byte_Size` is used to determine the size of the element to be allocated. This number is passed to the MTS subroutine `Getspace`. It allocates the requested amount of memory and returns a pointer to it, which is then assigned to the variable `New_Elem`. (Other methods of allocating list entries might be appropriate in some situations.)

The memory locations pointed at by `New_Elem` are referred to with an expression of the form `New_Elem@`. That is, the at sign is used to “follow” a pointer to the item it points to. Then the fields of the new item can be accessed with the usual “dot” syntax:

```
    New_Elem@.Symbol := Elem
```

would assign the value of the variable `Elem` to the field `Symbol` of the variable that `New_Elem` points to.

The element could be hooked into the list (at the front), by statements of the form

```
    New_Elem@.Next_Symbol := Symbol_Table;
    Symbol_Table := New_Elem
```

The first of these makes the new element point to the rest of the list, and the second points the head-of-list to the new element.

Notice the difference between

```
    Symbol_Table := New_Elem
```

and

```
    Symbol_Table@ := New_Elem@
```

The first of these is simply an assignment of the current value of the variable `New_Elem` to `Symbol_Table`. That is, it copies the pointer, so afterwards both pointers access the same memory location. The second, however, assigns the object pointed at by `New_Elem` to the location pointed at by `Symbol_Table`. That is, it copies the *record* of type `Symbol_Table_Element`. It would only be valid if you had previously set `Symbol_Table` to point to a suitable variable. (And it doesn't make much sense in this example!)

The built-in procedure `Address` can be used to get a pointer to a variable. For example, if for some reason you wished to make `New_Elem` point to the variable `Item`, you could use

```
    New_Elem := Address(Item)
```

This would only be valid if `Item` was a variable of type `Symbol_Element_Type`, so that `Address(Elem)` would be of type `pointer to Symbol_Element_Type`.

5. Procedure Types

A procedure type description is used to describe a procedure. The description specifies the names and types of the parameters and of the result, if any.

The declaration of `Print_Result` at lines 44–50 illustrates a procedure with four parameters and no result. When it is called, as at lines 101 and 105, the call must pass values that are compatible with the specified types of the parameters. Thus the first parameter must be a `Symbol` (`character(10)`), the second a `character(1 to 10)`, and so on. Since the procedure returns no result, its call must appear as a *statement* in the program—you can't use it as part of an expression.

a. Call-By-Value

By default, Plus passes parameters from the caller to the called procedure by copying the value into a local variable within the called procedure. Within `Print_Result`, therefore, the parameter names `Sym`, `Method`, etc., act just like local variables that have been preset to the values specified in the procedure call. This copying is a one-way process. If `Print_Result` were to assign a new value to one if its parameters, it would affect only the value of this local variable within `Print_Result`. It would *not* be reflected back to the calling procedure. So if `Print_Result` were to change the value of the parameter `Sym`, it would have *no effect* on the value of the variable `Test_Elem` that is passed in the calls from `Main`. This type of parameter passing is known as **call-by-value**.

b. Call-By-Reference

Sometimes, of course, you want a procedure to be able to change the value of one of the variables that is passed to it. This can be accomplished in either of two ways:

- •• You can pass a pointer to the variable instead of the actual variable. If `Print_Result` needed to be able to change its first parameter, you could change the procedure declaration to specify

  ```
  parameter Sym is pointer to Symbol
  ```

  and the call to

  ```
  Print_Result(Address(Test_Elem), ...)
  ```

  Then inside `Print_Result`, the identifier `Sym` would be a local variable whose value was a pointer to `Test_Elem`, so `Print_Result` could change `Test_Elem` indirectly with a statement such as

  ```
  Sym@ := ...
  ```

- •• You can specify in the procedure type that the parameter is to be a **reference parameter**. This is really equivalent to the first solution, but causes the compiler to automatically pass a pointer and dereference it at each use inside the called procedure. This form of parameter passing is commonly known as **call-by-reference**.

  The declaration of `Linearsearch` and `Binarysearch` at lines 52–57 specifies that the parameter `Accesses` is a `reference parameter`. Thus the calls at lines 100 and 104 pass the *address* of the variable `Accesses` (a local variable in `Main`, declared at line 94). The statements at lines 163 and 165 in `Linearsearch` that change the parameter `Accesses` follow the pointer and update the variable to which it points—i.e., the local variable `Accesses` in `Main` is what actually gets changed.

As a rule, using call-by-reference instead of call-by-value results in the procedure call being somewhat more efficient, because the caller doesn't have to do the work of copying the value. The difference may be significant when large data items (array, records, long character strings) are involved. Call-by-value also requires extra memory to hold the copy of such objects. On the other hand, call-by-reference results in the code inside the called procedure being rather *less* efficient, since it has to chase an extra level of indirection each time the parameter is used. With reference parameters, it may also be unclear to somebody reading the program that when the procedure is called, the parameter passed is subject to change. The use of explicit pointers and the `Address` makes this rather more obvious.

Generally, you should probably use call-by-value when the parameter is small (integer, short string, etc.), and there is no requirement for the called procedure to be able to "pass back" a changed value to the caller. You should use call-by-reference or explicit pointers when the parameter is large, or *is* used to pass back information.

c. Procedure Results

Procedures `Linearsearch` and `Binarysearch` also specify a result, with the phrase

```
result Position is Array_Index
```

Thus the procedures are used as *functions* within the caller (lines 100 and 104). They return a value of type `Array_Index` and may be used in any context appropriate to that type. In this case, the value returned is simply assigned to the variable `Pos`.

Within the procedure `Linearsearch`, `Position` acts like a local variable of type `Array_Index`. Whatever value was last assigned to this variable is returned as the result of the function. Alternatively, the return statement may directly specify a result to pass back, which then overrides the value of the result variable.

Any kind of data type can be returned as the result of a procedure. For example, `Getsym` passes back a character string. However, it probably isn't a very good idea to return very large data items as function results, because of the extra memory required, and the overhead of copying the result. Instead, you should use a reference parameter or explicit pointer parameter if you are writing a procedure that needs to set the value of a large variable in the caller.

d. Other Procedure Type Descriptions

Plus programs often need to call procedures that are written in other languages with different calling conventions. Thus Plus provides quite a few options for procedure type descriptions to allow accessing such routines. It is possible, for example, to make sure the call is compatible with the OS Type I ("Fortran") linkage, to specify that parameters are to be passed in registers, that some parameters are optional, to access the "return code" from a procedure and so forth. For all the details, see Chapter III.

e. Procedure Variables

Procedure types are most often used in a procedure declaration, but they are not limited to such use. You can also have procedure variables, arrays of procedures, record fields that are procedures, and so on.

The value assigned to a procedure variable must be a procedure name (as declared in a procedure declaration) or another procedure variable of the same type. You can also assign the value `Null` as a special "no value" indicator.

A procedure variable is called just like a procedure value. For example, if you declared

```
type Table_Search_Procedure is
      procedure
      parameter Element is Symbol,
      reference parameter Accesses is Integer
      result Position is Array_Index
      end;
procedures Linearsearch, Binarysearch are Table_Search_Procedure;
variable Search_Routine is Table_Search_Procedure
```

you could assign the variable a value:

```
Search_Routine := Linearsearch
```

and call the procedure assigned to `Search_Routine` as

```
Pos := Search_Routine(Test_Elem, Accesses)
```

Notice the difference between

```
Pos := Linearsearch(Elem, Accesses)
```

and

```
Search_Routine := Linearsearch
```

Since the first specifies parameters, it is a procedure call. Linearsearch is called and the result returned is assigned to Pos. The second does not specify parameters, so it is not a procedure call. It represents the procedure "value" which is then assigned directly to the variable `Search_Routine`. To call a procedure with no parameters you must still specify a parameter list consisting of just the parentheses `()`, in order to distinguish the call from an assignment of the procedure value.

It is sometimes useful to be able to pass the name of a procedure as a parameter to another procedure. If you wanted to generalize Example 2 to implement a procedure that could be used for testing other "table searching" procedures, you might declare

```
procedure Test_Search is
      procedure
      parameter Sym is Symbol,
         Search_Routine is Table_Search_Procedure,
         Method is character(1 to 10)
      end
```

Lines 99–105 of `Main` could be replaced with

```
Test_Search(Test_Elem, Linearsearch, "linear");
Test_Search(Test_Elem, Binarysearch, "binary")
```

and the definition of `Test_Search` would be simply

```
definition Test_Search
   variable Pos is Array_Index,
      Accesses is Integer;
   Pos := Search_Routine(Sym, Accesses);
```

```
      Print_Result(Sym, Method, Pos, Accesses)
   end Test_Search
```

M. Executable Statements

1. Assignment

A simple Plus assignment statement is of the form

```
Var := Expr
```

where `Var` is a variable to which a value is to be assigned, and `Expr` is an expression yielding the value.

You can assign any type of variable, not just simple values, in an assignment statement. An entire array or record can be copied with one assignment. But the two sides of the assignment must be of the same type.

The left-hand-side of the assignment may actually be any expression which results in a "name"—for example, a subscripted array, a field of a record, the location referenced by a pointer. We'll say more about names and values soon.

You can assign the same value to more than one variable in a single statement, by writing the left-hand-sides separated by commas. For example,

```
Low, High := 0
```

assigns zero to both variables. The right-hand-side of such a multiple assignment is evaluated once only.

Plus also lets you specify an operator in conjunction with assignment. The statement

```
Table_Size +:= 1
```

(line 68) is a shorthand for

```
Table_Size := Table_Size + 1
```

You can use similar notation for any of the binary operators `+`, `-`, `*`, `/`, `mod`, `||`, `|`, `&`, or `xor`, and for any left-hand-side expression.

The combination of the operator with the `:=` sometimes allows the compiler to generate better code. For example, in a statement like

```
Count(Pos + 1) +:= 1
```

(where `Count` is an array), the array subscript calculation only has to be done once.

If `Str` is a varying-length character string, then

```
Str ||:= " something"
```

is equivalent to

```
Str := Str || " something"
```

However, it appends the right-hand-side directly to the end of `Str`, without ever needing to compute the expression `Str || " something"`. As a matter of fact, the form

```
Str := Str || " something"
```

is not really correct Plus. For expressions involving character types, you should avoid using the left-hand-side variable as part of the right-hand-side. This is because the result is built directly in the left-hand-side variable. It happens to be harmless in this case, but for a statement like

```
Var1 := Var2 || Var1
```

the wrong result will occur, since execution will first move the value of `Var2` into `Var1`, and then access the wrong value when `Var1` is concatenated onto it.

At the moment, this situation is not usually detected by the compiler.

2. Expressions

Expressions in Plus are formed in the usual way, by combining various operands with appropriate operators and parentheses.

The primitive operands out of which you compose an expression include constants, symbolic constants, variable names and function names. The repertoire of operations you can use includes:

a. The usual arithmetic, bitstring and logical operators. For details, see Section I in Chapter III (page 83).

Expressions involving arithmetic operators follow normal precedence rules. That is,

```
A + B * C
```

is interpreted as if it were

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

Rather than introducing a complex precedence hierarchy, several other operators are given precedence equal to the arithmetic operators. The complete precedence hierarchy is given on page 84. You can always use parentheses to override the standard precedence or to clarify an expression.

b. Array subscripting, denoted by a parenthesized expression following the array name, as in:

```
Table(Table_Size)
```

c. Selection of a field of a record, denoted by the operator “`.`”. If `Elem` is of type

```
record
   Symbol is Symbol_Type,
   Reference_Count is Integer
end
```

then `Elem.Symbol` is the first field and `Elem.Reference_Count` is the second.

d. Procedure calling, indicated by a parenthesized list of parameters following the procedure name. For example

```
Linearsearch(Test_Elem, Accesses)
```

calls `Linearsearch`. A procedure with no parameters is called with an empty parameter list, as in

```
Getsym()
```

e. Following a pointer. The `@` operator is used to "dereference" a pointer to access the item that is pointed at. For example, if `Sym` is of type `pointer to Symbol`, then

```
Sym@
```

is the `Symbol` that it points to.

Plus strictly controls which operators may be applied to different types of operands. For example, `+` can be applied to numeric operands, but not to character ones. In a similar way, you can only use array subscripts for arrays, field selection for records, and so on. The result of each operation has a type which is derived from the type of the operand or operands and the operator used.

Notice that for each composite type there is a corresponding operation that accesses the element. The structure of a complex expression corresponds quite directly to the structure of a type. For example, if `Ptr` is of type

```
pointer to array (1 to 100) of
   record
      Field is
         procedure
         end,
      ...
   end
```

then `Ptr@` is an object of type

```
array (1 to 100) of
   record
     Field is
         procedure
         end,
      ...
   end
```

and `Ptr@(I)` is of type

```
record
   Field is
      procedure
      end,
   ...
end
```

and `Ptr@(I).Field` is of type

```
    procedure
    end
```

which is a procedure with no result, hence may be called as

```
    Ptr@(I).Field()
```

This is rather more complicated than the kind of types and expressions you are likely to use.

3. Names and Values

A **name** is an expression which corresponds to a memory location. The results of some expressions in Plus are names. Certain contexts in Plus (for example the left-hand-side of an assignment) require name expressions.

A **value** is a quantity that may be stored in a memory location. All constants are values, and the result of most expressions is a value. Operands for most operators must be values. If a name expression occurs in a context that requires a value, the compiler will always "dename" the expression and use the contents of the specified location as a value. The converse is *not* true—that is, if a value expression occurs in a context requiring a name, the compiler does not automatically generate a name.

The simplest name expressions are variable identifiers. The operations of subscripting and field selection, when applied to a name result in a name. When these operations are applied to a value, the result is a value. Similarly, the built-in procedure `Substring` results in a name if the first parameter is a name and the length of the substring is constant, and a value otherwise.

The dereferencing operator `@` takes a *value* of a pointer type, and converts it to a *name* of the resulting object type.

The built-in procedure `Address` takes a *name*-expression of any type as an argument and gives as its result a *value* which is a pointer to the argument.

The left-hand-side of an assignment must be a name. The parameter of `Address` must usually be a name. When a procedure parameter is defined as a reference parameter, the corresponding argument of a procedure call must be a name.

Plus provides an attribute `value` (see page 72) which may be specified for a type to indicate that an expression of the type may only be used in a "value" context. This is implemented by automatically "denaming" the name any time it is used in the program, so it won't be valid if the context requires a name. That means you can't store into it by using it on the left-hand-side of an assignment, or pass it to a procedure in such a way that the procedure could store into it.

If `@` is applied to a pointer whose object type has the `value` attribute (i.e, of type `pointer to value ...`), after dereferencing, the resulting name is immediately "denamed", again resulting in a value and so guaranteeing it cannnot be stored into.

`Address` may also be used with a constant as a parameter. This will result in a value which is a pointer to a *value*. (And hence, you can't use this pointer as an indirect way of corrupting the constant.) Similarly, a constant may be passed to a reference parameter, but only if the parameter type specifies `value`. Currently, `Address` and reference parameters cannot be used with any kind of value except constants.

The following examples may help clarify these interactions. Assume these declarations:

```
type T1 is array (1 to 100) of character(1);
variables Ind is (1 to 100),
   V1 is T1,
   V2 is character(1),
   P1 is pointer to T1,
   P2 is pointer to value character(1);

procedure Sub1 is
      procedure
      result R is T1
      end;

procedure Sub2 is
      procedure
      result R is pointer to T1
      end;
```

Then the statement

```
V2 := V1(5)
```

the subscripting operation results in a name (of type `character(1)`). This is then denamed automatically to obtain a value which is assigned to the name `V2`.

In the statement

```
V1 := Sub1()
```

`Sub1` is a procedure *constant*, and hence a value. The call results in a value of type `T1` which is then assigned to `V1`.

In

```
V1 := Sub1
```

the right-hand-side is a constant of a procedure type while the left-hand-side is a name of an array type; hence a type error message will be given. (The procedure `Sub1` is *not* automaticallly called.)

The statement

```
Sub1() := V1
```

is illegal, since the result of the call is a value.

In the statement

```
V1 := Sub2()@
```

the call results in a value of type `pointer to T1`. The dereferencing operator `@` then produces a name of type `T1`, which is then automatically denamed to obtain the value to assign to the name `V1`.

In this case,

```
Sub2()@ := V1
```

would be legal, since the left-hand-side results in a name. (`Sub2` returns a pointer to some memory location, then the value of `V1` is assigned to whatever location was returned.)

In

```
P2 := Address("X")
```

the call of `Address` results in a pointer to a value of type `character(1)`. This may then be assigned to `P2`, since it is defined as a pointer to a value.

```
P2 := Address(V2)
```

would also be legal. (A pointer to a name may be assigned to a pointer to value.)

```
V2 := P2@
```

would assign the character at which `P2` points, to the variable `V2`. The sequence of operations involved is: `P2` is denamed, resulting in a value of type `pointer to value character(1)`. This is then dereferenced resulting in a name which is immediately denamed because of the `value` attribute. It results in a *value* of type `character(1)`, which is then assigned to `V2`.

```
P2@ := "Z"
```

would be illegal; after the dereference, the left-hand-side is a *value* of type `character(1)`, hence an assignment to it is not allowed.

4. If Statements and Conditions

If statements allow you test expressions and choose between alternatives. So, in lines 195–204:

```
if Element < Table(Pos)
then
   High := Pos - 1
else
   Low := Pos + 1
end if
```

the value of `Element` is compared to `Table(Pos)`. If it is “less”, the statements in the then-part are executed; otherwise the statements in the else-part are executed. After either alternative, of course, execution continues with the statement following `end if`.

You can put an arbitrary list of statements (including other if statements, loops, etc.), in either part of the if statement. The keyword `else` and following statements may be omitted entirely if there is nothing to be done in that case. For example, the if statement at lines 79–83 has no else-part.

A sequence of nested if’s can be abbreviated using `elseif`, as in the if statement that runs from line 120 through 129. It is equivalent to the following pair of nested if statements:

```
if Return_Code ¬= 0
then
   Sym := ""
else
   if Length(Str) > Max_Sym_Length
   then
      Message(Msg, "Error - symbol too long</>");
      Sym := Substring(Str, 0, Max_Sym_Length)
   else
      Sym := Str
   end if
end if
```

The sequence of `... elseif ...` can be repeated many times.[4] Notice that when the `elseif` form is used, there is only one `end if` to terminate the whole `if... then... elseif... then... else... end if` construction.

A long chain of `elseif`'s can sometimes be replaced by a select statement. In such situations, the select statement will generally be much more efficient. The select statement is described in Chapter III, page 89.

The expression in an if statement must be one that evaluates to a numeric value. It is considered "true" if the value is *non-zero* and "false" if the value is *zero*. There is no built-in type "logical" or "Boolean" in Plus. The identifiers `True` and `False` are predefined as constants with the values 1 and 0 respectively.

Operators like `<`, `<=`, `=`, etc., compare the two operands specified, and result in a value of 1 if the specified relationship is true, and 0 if it isn't. So, in executing the above if statement, the expression `Return_Code ¬= 0`, is first evaluated. The result of this is an integer, either 0 or 1. This result is then tested for 0/non-0, and the if branches accordingly.

Compound conditions can be built up using `and` and `or`. For example, if you write

```
if I <= Max_Number_Symbols and Table(I) ¬= Test_Elem
then
   ...
end if
```

the then-part is executed only if *both* conditions are true. When you use such a compound condition, the second condition will be evaluated only if necessary. That is, if the first condition is false, then it doesn't matter what the value of the second condition is—the overall effect must be false. So Plus doesn't bother to evaluate the second condition. Another way to express this is that the if statement is evaluated as if you had written

```
if I <= Max_Number_Symbols
then
   if Table(I) ¬= Test_Elem
   then
      ...
```

[4] The current compilers limit it to a total of about 25.

```
   end if
end if
```

This form of evaluation is not only more efficient than evaluating both expressions; it also means you can use compound conditions where the second condition might be undefined or otherwise invalid if the first was false. In this example, if `I` is greater than `Max_Number_Symbols`, an array subscript error might arise if Plus attempted to evaluate `Table(I)`.

Analogous considerations apply to compound conditions using `or`. If the first condition is *true*, then the value of the second is irrelevant, so it isn't evaluated.

More complex compound conditions can be used, but if you mix `and`'s and `or`'s in an expression, you must parenthesize to make the order of evaluation clearer.

Conditions and compound conditions such as these are just expressions which evaluate to 0 or 1. They most often appear in the context of if statements, but they can be used in any appropriate context, such as assignment to a numeric variable. The standard Plus source library includes a definition for type `Boolean` as `(False to True)`— i.e, `(0 to 1)`. You might use this for "flag" variables in a program as in

```
variable Found is Boolean;
...
Found := I <= Max_Number_Symbols and Table(I) ¬= Test_Elem;
...
if Found
then
   ...
end if
```

5. Looping Statements

Plus provides two looping statements, the cycle statement and the do statement.

a. Cycle Statements

The cycle statement is very general. It just specifies that the statements between the keyword `cycle` and the matching `end cycle` (or just `end`) are to be be repeated indefinitely, until an exit statement inside the loop is performed. For example, lines 75–86 are repeated until either the exit at line 78, or the one at line 82 terminates it. In either case, execution continues after the end of the cycle (at line 88). The exit at line 78 specifies a condition; this exit terminates the loop only if the condition is true (non-zero). It is exactly equivalent to

```
if Length(Elem) = 0 or Elem = "\end"
then
   exit
end if
```

A cycle may also be terminated by executing a return statement in the loop, since that terminates the entire procedure containing the loop.

b. Do Statements

Since looping with an increasing or decreasing index is a very common situation, Plus provides a simple "do loop" to simplify writing such loops. Lines 164–167 provide an example. The statements between the heading `do ...` and the matching `end` are repeated, with the variable `Pos` assigned successive values. The loop terminates after it has been executed with the value of `Pos` equal to `Table_Size`. It may also terminate "early" by executing the return statement inside it. You can also terminate a do loop before the final value is reached by executing an exit statement.

A do loop of the form

```
do Index := Start_Value to End_Value
   /* statement list */
   ...
end do
```

is exactly equivalent to a cycle statement of the form

```
if Start_Value <= End_Value
then
   Index := Start_Value;
   cycle
      /* statement list */
      ...
      exit when Index = End_Value;
      Index +:= 1
   end cycle
end if
```

Note that after the loop finishes, the value of `Index` will be the value that it had the last time it executed, and that if `Start_Value` is bigger than `End_Value`, the body of the loop is never executed and the value of `Index` is unchanged.

Plus also allows a loop to "count down" by specifying `downto` instead of `to` in the loop heading.

Do loops always increment or decrement by one. Use the cycle statement if you require more general loop control.

6. Return Statements

The return statement is just used to terminate a procedure and go back to the caller. You can specify a return value as part of the statement, so

```
return with 0
```

at line 168 is equivalent to

```
Position := 0;
return
```

(`Position` is the identifier declared as the result.) You can also specify a condition, so

```
return when Table(Pos) = Elem with Pos
```

at line 166 is equivalent to

```
if Table(Pos) = Elem
then
   return with Pos
end if
```

If both `when` and `with` are used, they can occur in either order, so line 166 could also be written as

```
return with Pos when Table(Pos) = Elem
```

N. The Message Procedure

Example 2 illustrates simple use of a Plus library routine, `Message`, which produces formatted output. `Message` is *not* a part of the Plus language. It is a procedure, written entirely in Plus, that has been put into the standard Plus library because it has proven useful in many Plus programs.

Before using `Message` you must initialize it by calling the procedure `Message_Initialize`, as at line 67. `Message_Initialize` returns a pointer to a control block (of type `pointer to Stream_Type`) which `Message` uses to keep track of what it is doing. This pointer is passed as the first parameter in all calls to `Message`. You should not attempt to change anything inside the control block returned by `Message_Initialize`—it is entirely private to the message routines.

The second parameter to `Message` is a string to be emitted as the message. The message string may specify points at which values are to be substituted via codes surrounded with `<` and `>`. After the message string, there may be zero or more additional parameters, which are the values to be converted and substituted into the string. For example, at line 139, the call

```
Message(Msg, "<v> search: <v> ", Method, Sym)
```

emits the string as a message, with the value of `Method` substituted for the first `<v>` and the value of `Sym` substituted for the second. Here, `<v>` is a code for "varying string" and indicates the corresponding parameter is a Plus varying-length character string to be inserted in the message.

A message may be built up across a series of calls to the message routine. It is actually emitted only when the sequence `</>` is encountered in the string. Procedure `Print_Result` therefore prints only a one line message, created by three calls to the procedure (either lines 139, 142, and 146 or lines 139, 144, 146). It is terminated by the `</>` at line 146.

You can also emit more than one line in a single call to `Message`—each `</>` terminates a line and begins a new one. The two calls at line 70–71 could be replaced with a single call with one very long string for the second parameter.

The call to `Message_Terminate` at line 108 emits any incomplete messages, then releases the control block that was allocated by `Message_Initialize`.

There are a large number of codes that may be specified between `<` and `>`. These may specify insertion and various conversions to be applied to subsequent parameters, as well as various other operations such as emitting the line, tabbing to a specified position, padding the next parameter, and so on. Unfortunately, the message routine has no way of determining the types and size of the parameters to be substituted, so it is up to you to specify this in the substitution codes. This may require some intuition as to how Plus allocates variables. For example, the variable `Pos` is declared as `(0 to 600)`; this will be allocated as a halfword. The code `<hi>` used in line 120 means the corresponding parameter is a halfword integer, to be converted to an integer string. However, `Accesses` is declared as as `Integer` (= `(-2147483648 to 2147483647)`) which is allocated as a fullword integer. For this, the code `<i>` is used as at line 144. `Method` is declared as `character(1 to 10)`, which will be allocated as a one byte length field followed by the characters. For this, the code `<v>` is used.

There are other codes for one byte integers, varying strings with a halfword or fullword length field, fixed length strings, floating-point (of various lengths), hexadecimal conversion, and so on. Each code has a short (one or two character) form, and a longer, more mnemonic form. `<halfwordinteger>` and `<integer>` could have been used in place of `<hi>` and `<i>`.

By default, the message routine produces its output on `Sprint`. However, there are codes that can be used to direct the output to other output units or specific files and devices. You can set up an arbitrary number of independent output streams by making repeated calls to `Message_Initialize`.

For all the details see the writeup for the Message routine.

O. About the semicolons

In this section, we'll explain just when a semicolon is needed in a list of statements, and when it isn't. You don't really need to understand this completely, since the Plus compiler will usually let you get away with inserting unnecessary ones in "reasonable" places.

Plus follows the Algol tradition of using the semicolon as a *separator* between statements in a list of statements. To fully understand this, you must be aware of what constitutes a statement.

The program fragment

```
cycle
   Table(Table_Size) := Getsym();
   Table_Size +:= 1
end
```

contains a list of two assignment statements in the loop, so they are separated by a semicolon. `end`, however, is *not* a statement—it is just one of the punctuation marks that makes up the loop—so there's no need for a semicolon at the end of the second assignment. You may prefer to always put a semicolon there, however, so you don't have to remember to add it if you later add a third statement to the loop.

Similarly, `cycle` is not a statement by itself, so you don't follow it with a semicolon. And in an if statement like

```
if Rc ¬= 0
then
   Sym := ""
else
   Sym := Str
end if
```

you don't have to separate the keywords (`then` and `else`) from the statements in the then-part or else-part. (But again, when a *list* of statements appears in either alternative, the individual statements of the list must be separated by semicolons.)

Now, the entire cycle or if statement is *itself* a statement, so if this appears as part of a larger list of statements, it must be separated from its successor. Thus in

```
cycle
   Table(Table_Size) := Getsym();
   ...
   Table_Size +:= 1
end;
Sprint_String("Input Complete")
```

the semicolon after the `end` is required to separate the entire `cycle ... end` statement from the following `Sprint_String` procedure call.

P. The Rest of Plus

This chapter has covered the basic features of Plus in some detail. However, there is more to Plus that we haven't mentioned here. There are several additional statements, a large number of built-in procedures, and lots of additional options for types, declarations and procedure definitions.

All these are explained in the next chapter.

III. Language Details

This chapter presents a more advanced description of Plus. The description is quite informal, and relies a lot on examples.[1] However, it attempts to be accurate and complete. A complete BNF definition of the current syntax appears as Appendix B.

The Plus compilers are still under development. Some features of the language described herein are partially or totally unimplemented in some compilers. Restrictions and other properties of the current implementations of Plus are described throughout this chapter. We've attempted, however, to distinguish at all times between the design of the language and the status of its current implementations. Except as otherwise noted, everything in this chapter should apply to all compilers.

A. Program Format

Programs are completely free-format, with the restriction that a single lexeme cannot be split across two lines. Comments are surrounded by `/*` and `*/` as in PL/I or C, and may continue across an arbitrary number of lines. The semicolon is required as a *separator* between two statements in a list of statements. The syntax is fairly forgiving, and extra semicolons generally won't cause any problems.

Keywords of the language are reserved words. A complete list of the keywords appears as Appendix C.

Case is not significant in input to the compiler.

B. Compiler Input

Input to the compiler consists of a sequence of statements each of which may be a declaration, a global block, a procedure definition, or a compile-time statement. See Examples 1 and 2 in the previous chapter.

Declarations which are not contained in a procedure definition define global identifiers which may be referenced by all subsequent procedure definitions or statements. Declarations which are contained within a procedure definition are local to that procedure.

A procedure definition contains executable statements and declarations. Executable statements are allowed *only* within a procedure definition or inside a macro body.

Compile-time statements allow conditional compilation and a variety of other compile-time actions.

C. Compilation Structures

Plus probably departs furthest from its Pascal heritage in the area of compilation units and global variables. Pascal provides nested procedure definitions, with variables at one level accessible by all procedures nested within it. The problem with this approach is that it does not allow for separate compilation of the individual procedures—separately compilable procedures generally cannot share variables except via their parameter lists.

The approach taken by Plus is similar to that of C or Fortran. A program consists of a set of non-nested, separately-compilable procedures. Communication among procedures is by

[1] Some day, we'll add a more precise description of the syntax and semantics!

means of parameters and global variables. Global variables may be defined either by means of variable declarations which are external to all procedures, or by inclusion in a sort of "common area" called a **global block**. (Global blocks are implemented by the PDP-11 and System 370 compilers by using pseudo-register vectors. This allows the code to be completely reentrant and independent of operating system services. The Motorola 68000 compiler uses the application's global area on the Macintosh, and "bss" space on the AMIGA.)

1. Procedures

A procedure in Plus consists of two parts, a procedure declaration and a procedure definition. The **procedure declaration** specifies the type of the procedure. The **type** specifies the names and types of its parameters and return value (if any). The type may specify that some parameters are optional, and that others may be repeated an arbitrary number of times. It may also specify that parameters are to be passed in registers, and/or that the address of the parameter is to be passed rather than its value. The type may also request (via the keyword `system`) that calls to the procedure must conform to the standard linkage used in the operating system. See Section F–8 (page 71) for details of this attribute. See Section E–13 (page 64) for details of other aspects of the procedure type.

The procedure definition contains the series of **statements** to be executed when the procedure is called. The heading of the definition may specify that non-standard entry code is to be generated. The end of the definition is indicated by one of `end`, `end procedure`, `end definition`, or any of these followed by the name of the procedure.

The procedure declaration contains information that must be known to both the definition part and to any other procedure that wishes to call it.

The definition of a procedure must be preceded by its declaration. Any call of a procedure must be preceded by a declaration of the procedure called.

The declaration and definition of a procedure may be presented separately. For example:

```
procedure Print_Result is
      procedure
      parameter Sym is Symbol,
         Method is character(1 to 10),
         Pos is Array_Index,
         Accesses is Integer
      end;
   ...
/* other declarations, globals, definitions,
   etc. */
   ...
definition Print_Result
  ...
end Print_Result
```

Alternatively, the declaration and definition may be combined in a single construct of the form

```
procedure Print_Result is
      procedure
      parameter Sym is Symbol,
         Method is character(1 to 10),
         Pos is Array_Index,
         Accesses is Integer
      end
definition
  ...
end Print_Result
```

When the procedure declaration and definition are combined, the procedure identifier is not repeated following the keyword `definition`.

Procedure declarations and definitions may be presented separately either to facilitate separate compilation or to permit circular calling sequences. In the case of separate compilation, note that each compilation which contains a call to a procedure must contain a declaration for that procedure.

2. Global Variables and Global Blocks

Global variables may be accessed by any procedure provided the appropriate declarations are present. A variable may be made global in either of two ways:

a. by placing the variable declaration inside a **global block**. This is the preferred method when there are a number of global variables, since it reduces the run-time register requirements;

b. by placing the variable declaration outside of any procedure declaration. Such a global variable acts exactly as if it were in a (nameless) global block by itself.

A global block may contain any of the declarative statements of the language described in Section G (see page 73). A global block may appear in the compiler input either outside of any procedure (in which case the definitions it contains remain for all following procedures) or internally to a procedure (in which case it is discarded at the end of the scope block in which it occurs). There is no limit on the number of global blocks in a program. However, code quality may suffer somewhat if a single procedure references variables from a large number of separate global blocks.

The heading of a global block may specify an external symbol to be used for the global area instead of the default symbol generated from the global block's identifier. The end of a global block is indicated by `end` or `end global`, or either followed by the name of the global block.

At execution time, all procedures access the same copy of any global variable, regardless of where the declaration occurs.

On the System 370 and PDP-11 the code is kept fully reentrant by using pseudo-registers to implement global variables. Each global block will be one pseudo-register; individual definitions within global blocks will not generate external symbols. A variable that is neither inside a procedure nor inside a global block is a separate pseudo-register.

On the Macintosh, each global block or global variable is allocated space in the application's global storage area, addressed from `A5`.

On the AMIGA, each global block or global variable defines a separate “bss hunk”.

3. Global Environments

Normally, the global storage accessible consists of all global variables defined in the program, and remains “fixed” throughout execution of a program.

However, Plus provides a way for a family of procedures to have its own global storage that is independent of the global storage used in the rest of the program. Switching from one global storage to another can be performed at the time of a procedure call, either by the caller, or by the entry sequence of the called procedure. Plus implements the concept of a **global environment** to support switching global storage.

Note

Global environments are an “advanced topic”. Most programmers should not need to be aware of the complications described below.

System 370 Note

This facility in Plus/370 matches that of the MTS “Coding Conventions”, but implements the additional mechanism of switching during the entry sequence of the called procedure.

Implementation Restriction (PDP-11)

Plus-11 does not support the mechanisms for switching global storage environments.

Motorola 68000 Note

Plus/68000 supports the switching of global storage environments, but this is probably only useful on the AMIGA, when calling “system library” procedures.

Every Plus procedure has an associated **environment type**. An environment type may be either a special type `global(`*n*`)`, where *n* is a `bit(32)` constant, or it may be a pointer to a record type. It may also be specified as `unknown`. The environment for a procedure is specified with the environment attribute; see Section F–2, page 69.

An environment of type `global(`*n*`)` means that the global storage is defined by the usual method of defining global storage. A value of such a type is just the base address of a region of storage allocated for the global variables. All global variables declared in an input file to the compiler are considered to be part of one global environment, of type `global(%Global_Id)`, where `%Global_Id` is a settable compiler option (it defaults to `"PLUS"`).

Implementation Restriction (System 370)

In Plus/370, a pointer to a record type can only be used for an environment if the initial portion of the record contains certain reserved fields and is initialized appropriately as required by the MTS “Coding Conventions”. This is described in Appendix D.

One procedure may call another only if either a) the caller's environment type is compatible with the environment type of the called procedures, or b) the caller provides a value of a compatible environment type as part of the procedure call.

Examples:

```
procedure P1 is environment global("QQSV")
          procedure
          ...
          end,
     P2 is environment global("FOO")
          procedure
          ...
          end,
     P3 is environment pointer to Rec_Type
          procedure
          ...
          end,
     P4 is
          procedure
          ...
          end,
     P5 is environment unknown
          procedure
          ...
          end
```

Given the above, any of `P1`, `P2`, `P3` or `P4` could call `P5` and vice-versa.

`P1` could not call `P2` (or vice-versa) unless it provided an appropriate environment value to switch to, since they have different global types for environments. Similarly, `P1` or `P2` could not call `P3` (or vice-versa) unless they provided an appropriate environment value to switch to. Section K–3, page 87 describes the syntax used to switch environments at the time of a call.

The environment of `P4` is the default, which is `global(%Global_Id)`. Thus, `P1` could call `P4` directly if and only if `%Global_Id` had the value `"QQSV"`.

The entry sequence of a procedure may also switch environments by specifying a new environment as part of the procedure heading, as described by Section P–2, page 98. When this is used, the caller must *still* call with the appropriate environment type. This environment is in effect for the evaluation of the expression in the entry code which loads the new environment. The new environment will be in effect for any calls from within the procedures, so will be used in determining compatibility of subsequent calls.

A Plus procedure can reference Plus global variables only if it is executing with environment `global(%Global_Id)`; otherwise the global variables are hidden inside the procedure, since they are part of a different environment. If the environment attribute is not given, `global(%Global_Id)` is assumed, so by default all procedures can access global variables.

Note that all global variables in a given compiler run are part of *one* global type. You can specify the name of that environment, but can't have some parts of the program use one and some parts use another. It *is* possible to have separately compiled pieces of the program use different global types.

Implementation Restriction (System 370)

MTS currently only provides rudimentary support for loading programs that use more than one independent PRV. Generally, this is only practical with separately loaded components, so it is mainly used with pre-loaded subroutine packages.

Implementation Restriction (Motorola 68000)

The Macintosh system and application structure effectively prohibit independent global storages, and neither the MPW nor MDS linkers have any support for them. Plus/68000does implement the use of environments which are pointers to record types.

If the procedure environment is defined by a pointer to a record, Plus global variables are not accessible inside the procedure. However, in this case the fields of the record type will be made accessible inside the procedure as if they were global variables; i.e., they may be referenced without qualification.

If a procedure also has a special linkage option, the parameters in the prologue that are passed to the linkage routine include the size of the environment and, for environments of type `global(`*n*`)`, the value of the constant *n*.

The predefined register variable `Environment_Base_Register` always has the same type as the current environment and may be used if necessary to access the environment value. However, it may be used for setting the environment *only* in a routine that has the `linkage none` option (and then, only by experts). The code generated by the Plus compiler assumes the environment is changed only as allowed by the procedure call and entry sequence options, and changes made at other times may not work as "expected".

4. External Variables

A variable may be declared **external** (see page 75). In this case the compiler will access it through an external symbol reference. It will not allocate storage for the variable, either as a local or global variable.

External variables must be defined at load-time either by methods outside of the Plus language (e.g., assembler), or by the use of an **entry constant**.

System 370 Note

External variables can be used to access data in Fortran common blocks on the System 370. To do so, the external variable would be declared as a record whose fields correspond to the variables of the Fortran common block.

Motorola 68000 Note

External variables are currently assumed to be in the global data area. This is largely so that they can be defined by an entry constant.

5. Entry Constants

A constant declaration in Plus may include the specification `entry` (see page 74), which causes generation of an object module containing the value of the constant. This is normally used with constant arrays and records to generate tables etc. The constant may then be referenced from other components or other languages by means of an appropriate external declaration.

Implementation Restriction (Motorola 68000)

When `%Target_Operating_System` has the value `"MAC/MDS"`, entry constants are not implemented because the basic MDS linker does not have the mechanisms to initialize data areas.

6. External Symbols

Each procedure, global block, external variable, each global variable which is not a member of a global block, and each entry constant, requires an “external symbol”. Individual variables within a global block do not require external symbols.

External symbols must obey restrictions imposed by the system linker. In particular, the MTS loader and `*Link11` require that all external symbols be at most eight characters long. The Macintosh and AMIGA loaders do not impose such a restriction.

The external symbol to be used may be specified by a string constant in the declaration of a procedure or an external variable, or in the heading of a global block. If an external symbol is not explicitly given, then the Plus identifier is used. If this is longer than the system linker allows, the compiler will form an external symbol. Plus/370 and Plus-11 take the first four and last four characters of the identifier. The compilers will check, within a single run, that any such generated symbols are unique; i.e, do not conflict with other external symbols. It is unable to check across separately compiled portions of a program.

Note that the external symbol for a global variable which is not part of a global block is always obtained from its identifier (if the `external` specification is used, the variable becomes an external variable, not a Plus global variable). To specify the external symbol, the variable must be enclosed in a global block.

Example:

```
global Global_One
   variable V1 is (1 to 100);
   ...
end Global_One;

variable External_One is (1 to 100);

variable Caseconv is character(256) external;

procedure Proc1;
procedure Proc2 external "P2";
```

```
definition Proc1
   global Global_Two external "G2"
      variable V2 is (1 to 100)
   end Global_Two;
   ...
end Proc1;

definition Proc2
   ...
end Proc2;
```

Using Plus/370 or Plus-11 in the above examples, `Global_One` is an external global with the external name `GLOB_ONE`. The variables declared within it may be referenced within any procedure that follows. These variables are *not* external symbols. `External_One` is a global variable, with the external name `EXTE_ONE` which may be referenced anywhere in the following procedures. `Caseconv` is an external variable which must be defined outside of Plus. Its external name is `CASECONV`; an alternate external symbol could be specified by a string constant following the keyword `external`. `Global_Two` has external name `G2`. The variables declared within it may be referenced only within procedure `Proc1`, unless the definition of `Global_Two` is repeated elsewhere. `Proc1` has external name `PROC1`, and `Proc2` has external name `P2`.

The only differences using Plus/68000 are that `Global_One` has the external name `GLOBAL_ONE` and `External_One` has the external name `EXTERNAL_ONE`.

7. Macros

Plus currently does not provide "internal procedures" as such. However, macros are provided to handle some of the situations where internal procedures might be useful. A macro associates a name with a piece of program text. The text is then substituted into the program whenever the name of the macro is subsequently encountered in an executable statement. Macros may have parameters, with the text given as the argument when the macro is invoked being substituted for the parameter name in the macro body.

Macro substitution is at the "lexeme" level. That is, the macro body or macro argument is interpreted as a sequence of tokens (keywords, identifiers, symbols), before any substitution occurs. The sequence of tokens is substituted where the name of the macro or a macro parameter occurs as an identifier.

The **body** of the macro may be either of two syntactic constructs—a parenthesized expression, or a **scope block**. (Basically, a scope block is a statement or sequence of statements.) A macro may only be invoked in places where the body is syntactically valid.

Macros are generally used for one of three reasons. They may be used to avoid the overhead of a procedure call for small sequences of code required in several places. They are convenient for defining interface code to non-Plus procedures, which may require the use of `Inline`, and/or type cheating of parameters. They are also useful for top-down programming, to allow a program to use a name for an action that will be defined separately.

Macros may be defined either inside or outside of a procedure. Each identifier used within a macro is normally associated with the definition in effect at the point where the macro is defined. It is possible, however, for a macro to have "free variables", which are associated at expansion time. (Any identifier which is used in a macro, but is not defined at the point where the macro is defined, is associated at expansion time.)

Further details and examples of macros are given in Section Q, page 99.

Note

Macros as described above may be removed from a future version of Plus in favour of internal or "inline" procedures. We recommend that macros be used only in ways that are compatible with procedures.

D. Identifiers

An identifier in Plus is a sequence of up to 100 characters, which may be letters, digits, or the characters `$`, `#`, or `_`. The first character may not be a digit. Upper or lower case letters may be used, but are considered equivalent. Thus the identifier `FALSE` is the same as the identifier `FaLsE`.

1. Uses of Identifiers

Identifiers are used in Plus for the purposes listed below. Each type of identifier is described in more detail elsewhere.

a. **Procedure Names**—Procedure names are specified in the procedure declarations and in the procedure definition if it occurs separately. They are used to invoke the procedure. See Section C–6, page 49 for restrictions on identifiers used as procedure names.

b. **Global Block Names**—Global block names appear only in the heading of the global block. Again, see restrictions in Section C–6, page 49.

c. **Macro Names**—Macro names appear in macro definitions and are used to invoke the macro.

d. **Symbolic Constants**—Symbolic constants are defined explicitly by means of the constant declaration, or implicitly by occurrence of the identifier in the list of an "identifier-list type" definition, described in section E–4, page 57.

e. **Type Identifiers**—Types may be given names in a type declaration. These names may then be used in any other situation requiring a type description.

f. **Variable Names**—Variables are declared with the variable and equate statements. Each variable is associated with a type by its declaration.

g. **Procedure Parameters and Results**—Procedure parameters and results are given names as part of the procedure type. Each parameter and/or the result is given a type, and is treated as a variable of that type within the procedure definition.

h. **Macro Parameters**—Macro parameters are defined in the heading of the macro, but are not associated with types. Macro parameters may be replaced by expressions of any type at macro expansion time. The specified expressions must be type-compatible with whatever context the associated parameters are used in.

i. **Record Fields**—The definition of a record type associates an identifier with each field of the record. The record field name is used to qualify the name of a variable of the record type, in accessing the particular field.

j. **Exit Labels**—Certain constructs in Plus may be labelled by preceding and following them with an identifier enclosed in the symbols `<` and `>`. These labels are used to designate the points to which "escapes" may be made from within the construct.

k. **Compiler Variables**—Compiler variables are special predefined identifiers which are used to set and test various compiler options. They always begin with `%`.

l. **Compiler Procedures**—Compiler procedures, like compiler variables, are identifiers beginning with `%`, and are used to invoke special compile-time actions of the compiler.

2. Definition of Identifiers

Every identifier used in a program must be defined, usually by an appropriate declarative statement. There are a few built-in procedures and constants which are predefined identifiers.

With one exception, any identifier must be defined *before* it is first used.

The one exception is that a pointer type description may refer to an undefined identifier as its object type. This allows for circular definitions in record types—e.g., a record of type `T1` may have a field which points to an object of type `T2`, which in turn may contain a pointer to another object of type `T1`.

The type of the object type identifier must be defined before any executable statement which accesses the object of the pointer. However, if no statement within the compilation manipulates the object (i.e., dereferences the pointer), the object type is allowed to remain undefined. (This provides an aid to separate compilation, since a separately compiled procedure need only include declarations for those objects which it manipulates, even if it references structures containing pointers to other objects.)

Examples:

```
variables V1, V2 are pointer to Undef;
    ...
V1 := V2;
V1@ := 5;
    ...
type Undef is (1 to 100);
V2@ := 5;
```

The assignment of `V1` to `V2` is valid, since it does not access the object of the pointer. The dereferencing operator `@` (at sign) is used to access the object pointed at by its operand; hence the assignment to `V1@` will result in an error message because the object of `V1` is of type `Undef` which is not yet defined. The assignment to `V2@` is valid, since the object type is defined previously.

3. Scope of Identifiers

Identifiers obey **scope rules** like those of Algol or PL/I. Identifiers declared in one scope can be referenced in any scope nested within it, unless the same identifier is declared in a nested scope. Identifiers may not be referenced outside of the scope in which they are declared. Variables declared within a scope *do not exist* outside of that scope.

The statement list inside any "bracketed" control structure forms a separate scope in Plus. Extra `begin...end` blocks are not required. A scope may contain declarations and executable statements intermixed.

The use of undefined identifiers as pointer object types interacts with the scope rules in the following way. If an undefined identifier occurs as the object of a pointer type in one scope, it is assumed to be implicitly defined in that scope. If it is subsequently used as the object of a pointer within a nested scope, the second use will be assumed to refer to the same type. In this situation, it will be invalid to define the identifier within the nested scope (which would cause the pointer type in the outer scope to refer to a type defined in the inner scope).

For example, in the sequence

```
variable V1 is pointer to Undef;
begin
   type Undef is ...
   variable V2 is pointer to Undef;
      ...
end
```

the definition of `Undef` within the begin block defines a new type. Therefore, the variables `V1` and `V2` are of different types. On the other hand, in the sequence

```
variable V1 is pointer to Undef;
begin
   variable V2 is pointer to Undef;
      ...
end
```

the use of the symbol `Undef` in the begin block is assumed to be the same as the use that is implicitly defined in the outer scope, and hence the two variables are of the same type. If a subsequent statement within the begin block attempts to define `Undef` (as a type, or as anything else), an error message will be issued.

E. Type Descriptions

A **type** is a description of the values which may be assigned to variables of that type. There are certain basic **scalar** types in the language, and rules for constructing more complex types like arrays and records out of basic types.

Type descriptions may appear in several contexts in the language. The most important contexts are variable and type declarations and in the descriptions of more complex types.

A **type declaration** simply associates an identifier with a type description. Thereafter, the identifier may be used in other type definitions in any context. The ability to give a name to a type allows you to define a type in one place and then use it elsewhere without further concern for the details of its representation.

For each scalar type, Plus provides a way of expressing constants of the type. For each type, certain operations are allowed. Every expression has a type, derived from the types of the operands and the operator involved.

The following sections describe the types provided and applicable operations.

1. Numeric Types

A numeric type is a type whose values may be integers in a given range.

Examples:

```
type Number is (0 to 32767);
type MTS_Line_Number_Type is (-99999999 to 99999999);
/* False and True are predefined constants. */
type Boolean is (False to True)
```

The operations defined for this type are

a. arithmetic operators `+`, `-`, `*`, `/` and `mod`,[2] and unary operations `+`, `-` and `abs`.

b. relational operators `<`, `<=`, `>`, `>=`, `=` and `¬=`, which perform an arithmetic compare, giving a result of 0 (false) or 1 (true).

Any numeric type is compatible, for assignment and for all the above operations, with any other numeric type or with certain bit types. The compiler will optionally provide run-time range checking to detect assignments of values out of the declared range.

2. Character Types

A character type is a type whose values may be character strings. There are two kinds of character types, fixed length types and varying-length types. A varying-length character type is expressed by giving the range of lengths that assigned strings may be. (This information may be used for run-time checking, and also sometimes allows the compiler to generate better code.)

The maximum length of a varying character type is used in allocating storage for a variable of that type.

Examples:

```
/* Note Standard_String_Length and Max_Symbol_Length are
   constants. */
type Fixed_String is character(Standard_String_Length);
type Symbol is character(0 to Max_Sym_Length)
```

Operations allowed for character types are concatenation (denoted by `||`), and the relational (comparison) operators. There is also a built-in procedure `Substring` which selects substrings of character names or values, and a built-in procedure `Length` which returns the length of a character value.

[2] Note `mod` is an operator, not a built-in function. Thus it is used in an expression as `X mod Y`, not as `mod(X,Y)`, as Fortran programmers might expect.

Character types are compatible, for the purposes of assignment and the above operations, with other character types, even of different lengths. Strings are *never* extended (with blanks or anything else) during operations. Character types are also compatible with certain bit types.

The length assigned by an assignment statement is always determined from the source (right-hand-side). It is an error to attempt to assign a value that is too long for the destination. The compiler will optionally generate code to test at run-time for invalid string-lengths that cannot be detected at compile time.

Character comparison is done lexicographically. That is, `"A" < "AB" < "B" < "BB"`. For strings of the same length, this is exactly what results for the System 370 from a CLC operation. For strings of different lengths the number of characters of the shorter are compared first. If these are equal, then the shorter string is considered less than the longer.

Implementation Restriction (System 370)

The current implementation cheats slightly on this definition by comparing the strings as if the shorter were padded with (binary) zeros to the required length. (Thus if one string is longer than the other, but ends in zeros, the strings may be found equal, although the shorter might be less than the longer according to the definition.)

Examples:

```
String := "";
String ||:= Integer_To_Varying(Count,0) || " records"
```

Note that `||:=` has the effect of appending the right-hand-side to the left-hand-side, provided the destination is a varying length string.

3. Bit Types

Bit types are a machine-oriented type that allows specifying storage allocation in terms of a fixed number of bits. For example:

```
type Machine_Address is bit(24)
```

Bit types will be coerced when necessary to other scalar types, so that bit values can be used to express other types in a machine-dependent way.

Plus distinguishes two kinds of bit types, right-justifying (or “index-like”) bit types and left-justifying (or “string-like”) bit types. The distinction is important when bit-types of different lengths are mixed in expressions, or when bit-types are mixed in expressions with other scalar types. The distinction is usually based on the word-size of the object machine. In the following discussion, `Word_Size` is 32 for the 370 and 68000 compilers and 16 for the PDP-11 compiler.

A bit-type is usually interpreted as right-justifying if its length is less than or equal to `Word_Size`, and as left-justifying if its length is greater than `Word_Size`. The attribute `left` may be used in a type description to force a short bit-type to be treated as left-justifying. The current implementations require that a right-justifying bit-type have length `<= Word_Size`. The 370 and 68000 implementations further require that a right-justifying bit type must be contained within four or fewer bytes. The PDP-11 compiler requires that it be contained within a word (i.e., it may not cross a word boundary). A left-justifying bit-type must have a length which is a multiple of 8 bits, and must be allocated at a byte boundary.

Right-justifying bit-types are compatible with any index-type (defined on page 58), including other right-justifying bit-types of different lengths. They are also compatible with left-justifying bit-types of the same length. Right-justifying bit-type values will be coerced to other index types if used as operands of operators requiring another type. Note that in the coercion to an index type, some right-justifying bit-types are treated as signed and some are not, depending on the actual bit length. It is up to the particular implementation to determine which bit lengths will be signed and which will not.

A left-justifying bit-type is compatible with any character type, with other left-justifying bit-types (of any length) and with right-justifying bit-types of the same length. Left-justifying bit types of length n behave similarly to character-types of length $n/8$.

The logical operators `|`, `&`, `xor` and ¬ are defined for compatible bit-types. (That is, for types of the same justification or the same length.) Index types will be coerced to right-justifying bit-types (of length `Word_Size`), and character types will be coerced to left-justifying bit-types if they are used as operands of these operators. When the operators are applied to a pair of left-justifying operands, the bit-strings are aligned at the left end, and the length of the result is the length of the shorter. When applied to a pair of right-justifying operands, the right ends are aligned, and the result is always a bit string of length `Word_Size`. When one operand is left-justifying and the other right-justifying (in which case the lengths must be the same), the result is a right-justifying `bit(Word_Size)`.

When comparison operators `<`, `<=`, `>` and `>=` are used to compare two bit-types, an arithmetic comparison will be performed if the types are right-justifying or of opposite justification and a logical comparison if they are left-justifying. The `Left_Justify` built-in function can be used to coerce a right-justifying operand into a left-justifying expression.

System 370 Note

`Word_Size` is 32 for Plus/370 and `bit(16)` and `bit(32)` are signed. A right-justifying bit type must be contained within four or fewer bytes.

PDP-11 Note

`Word_Size` is 16 for Plus-11 and `bit(16)` is signed. A right-justifying bit type must be contained within a word (i.e., it may not cross a word boundary). `|`, `&` and ¬ are not implemented for left-justifying bit types (except for constant expressions).

Motorola 68000 Note

`Word_Size` is 32 for Plus/68000 and `bit(16)` and `bit(32)` are signed. A right-justifying bit type must be contained within four or fewer bytes.

4. Identifier-List Types

The identifier-list type allows you to create new basic types by enumerating a list of identifiers which are to be the elements of the type.

Example:

```
type Device_Type is (Printer, Reader, Punch, Tape_Drive, Disk_Drive,
   Terminal)
```

The elements of the identifier list are automatically declared to be symbolic constants of the given type (and must therefore not be previously declared in the same scope).

The compiler is free to choose an appropriate internal representation for each element of the type. In fact, the representation used will be successive integers, starting with zero, but you cannot make use of this fact except by type-cheating.

The relational operators are defined for identifier-list types, with the values considered ordered as they appear in the identifier list (the first is smallest). Values of an identifier-list type are compatible only with other values of the same type, or with right-justifying bit types.

5. Real Types

Real types are used for floating-point numbers. The type definition specifies the number of decimal digits of precision wanted.

Examples:

```
type Short_Real is real(7),
   Long_Real is real(16);
variable V1 is real(5),
   V2 is Long_Real
```

Real types are compatible with bit types of appropriate size.

System 370 Note

For the 370 implementation, the precision n in `real(n)` is interpreted as

$1 <= n <= 7$ results in 370 single precision (4 bytes)
$8 <= n <= 16$ results in 370 double precision (8 bytes)
$17 <= n <= 34$ results in 370 extended precision (16 bytes)

Currently, real variables of different sizes cannot be mixed, even across assignment, so it is necessary to use type cheating or `Inline` to assign from a real of one size to a real of another.

Currently, there are *no* operations implemented for real types, except assignment. Comparison operations may be used, but will perform a logical comparison (string comparison), not a floating point comparison.

Implementation Restriction (PDP-11)

Real types are not implemented for Plus-11.

Implementation Restriction (Motorola 68000)

> Real types are not implemented for Plus/68000.

6. Index Types

A certain subset of the preceding scalar types are known as **index types**. Index types may be used for control variables in do loops, for subscripting arrays, and in certain other contexts. The index types include all numeric types, identifier-list types, `character(1)` types, and right-justifying bit-types.

The built in procedures `Low_Value`, `High_Value`, `Successor`, `Predecessor`, `Min`, and `Max` are defined for any index type.

7. Subrange Types

Any subrange of an index type is itself an index type. Subranges are indicated by giving the lowest and highest values of the type. (In fact, any numeric type is really a subrange of a predefined, unspecifiable type 'integer'.)

Subrange types allow the same operations as their "base type". Any subrange is compatible with any other subrange of the same type.

Examples:

```
/* Following is a subrange of Device_Type */
type Unit_Record_Type is (Reader to Punch);

/* Following is a subrange of type character(1). */
type Digit is ("0" to "9")
```

8. Set Types

Implementation Restriction (all compilers)

> Set types and all related operations are currently not implemented.

Set types allow defining variables whose values may be arbitrary *sets* of values from a given index type. Sets provide a very convenient way of expressing some programming constructs that in other languages would have to be represented by arrays of Booleans or bit strings.

Example:

```
type Mts_Modifiers_Type is (Indexed, Binary, Carriage_Control, Prefix,
      Peel, Machine_Cc, Trim, Special, Ic, Case_Conversion);
variables Required, Excluded are set of Mts_Modifiers_Type
```

A variable of type `Mts_Modifiers_Type` can be assigned a set of values; e.g.,

```
    Required := {Indexed, Trim};
    Excluded := {Carriage_Control, Case_Conversion}
```

Theoretically, any index type can be used as the base of a set, although there will be some implementation restriction on the possible size of the range. Sets are implemented using bit strings. The presence of an element in the set is indicated by an on-bit.

The set braces `{` and `}` (the alternative notation `(|` and `|)` may be used for devices which have no left-brace and right-brace) allow construction of sets. The operators `|`, `&` and `-` are defined to mean set union, intersection, and difference when applied to set types. The relational operator `subset` can be used to test whether one set value is a subset of another. The relational operator `in` can be used to test whether a particular value of the base type is in a given set.

Values of two set types are type-compatible if their "base types" are compatible. That is, `set of (1 to 10)` is compatible with `set of (5 to 20)`. The result of a set operation on these two might be of type `set of (1 to 20)`.

A value of a scalar type will be coerced into a set containing only that value when context requires it. For example:

```
    Required |:= Ic
```

is equivalent to

```
    Required |:= {Ic}
```

which means

```
    Required := Required | {Ic}
```

This therefore has the effect of adding the value `Ic` to the set `Required`.

A type-identifier for an index type may be used in a context requiring a set of that type, and is equivalent to the set containing all values of the index type.

Sets frequently allow the construction of efficient algorithms which would be difficult to do in most high-level languages. The above example indicates how a concept similar to the MTS I/O modifier pairs might be expressed in this language. Instead of using adjacent pairs of bits for modifiers, two sets are used. The set `Required` specifies those options which have been selected (e.g, `@Ic` results in `Ic` being placed in set `Required`). The set `Excluded` specifies those modifiers which are specifically not to be applied (e.g, `@¬Ic` results in `Ic` being placed in set `Excluded`). With this sense of modifiers, the modifier amalgamation algorithm required to combine the Fdname modifiers[3] with the operation modifiers can be expressed as:

```
    Combined_Required := (Fdname_Required - Op_Excluded) | Op_Required;
    Combined_Excluded := (Fdname_Excluded - Op_Required) | Op_Excluded
```

This will generate code that is very nearly as good as that in the assembler version.

[3] Under MTS, I/O modifiers may be specified as part of a "file or device name" to apply to all operations on that Fdname, and may also specified on each I/O operation. At each level, the modifier may be asserted as "on" or "off" or defaulted. In case of conflict, the operation modifiers have precedence over the Fdname modifiers.

As a final example, note the following is allowed:

```
if Device in {Reader, Printer}
then
   ...
end;
```

This means the same as

```
if Device = Reader or Device = Printer
then
   ...
end
```

but the first will generate better code, and is probably at least as easily understood.

9. Array Types

An array type is constructed out of two other types, an index type (which defines the type and range of the subscripts allowed) and an arbitrary type which defines the type of the elements. Note that any index type is allowed as the subscript type. It is possible to have arrays indexed by character, or by identifier-list types, as well as by numbers.

Examples:

```
variable Translate_Table is array character(1) of character(1);
type Symbol_Array is array (1 to Max_Number_Symbols) of Symbol
```

The elements of an array may be of any type, including another array type. Thus multi-dimensional arrays can be constructed out of arrays of arrays. Note that there is no way to define an array whose size is determined at run-time (but see Chapter VIII, page 140).

The only operations that can be performed on arrays are assignment, subscripting and comparison. An array of a given type can only be assigned to an array of the same type. Subscripting is denoted in the usual way, by means of a parenthesized expression following the array name. The subscript expression must be type-compatible with the specified index type of the array, e.g.

```
Translate_Table("a") := "A"
```

Two arrays of the same type can be compared, using the operators `=` and `¬=` only. However, some caution is required when comparing entire arrays. In some cases, the allocation of an array may require padding elements to maintain alignment requirements. When arrays are compared with a single comparison, this padding will be included in the locations compared. The result of the comparison may then be incorrect, since the padding bytes are likely uninitialized. This situation is not detected by the compiler.

For convenience in accessing elements of arrays of arrays, multiple levels of parenthesized expressions may be condensed into an expression list. For example, given the declaration

```
variable Matrix is array (1 to 10) of array (10 to 20) of Number
```

the `I,J`'th element may be referred to as either

```
Matrix(I)(J)
```

or

```
Matrix(I,J)
```

When a constant subscript is applied to a constant array, the result is a constant which may be used in any context requiring a constant. A variable subscript applied to a constant array does not result in a constant, since it requires a run-time calculation. Hence it cannot be used in contexts requiring a constant.

10. Pointer Types

Pointers in Plus must usually be defined in terms of the type of object that they point to. This allows full checking of the types resulting from use of the pointers. Given any type, `pointer to` that type is another valid type. The values of the pointer type are addresses of variables of the object type. It is also possible to have a pointer to a constant. In this case the value will be the address of a location containing that constant. See Section H, page 81.

Example:

```
variables First_Elem, Last_Elem are pointer to Symbol_Table_Element
```

The suffix operator `@` may be used to follow (or "dereference") a pointer. The result of applying this operator is a name of an element of the given object type. (If the pointer is a pointer to a value, `@` results in a value, not a name.)

A pointer to a variable is created by means of the built-in procedure `Address`. The argument of `Address` must be a *name* or a *constant* of any type. The result is a value of type `pointer to` the type of the argument. Thus,

```
variable Item is Symbol_Table_Element,
   First_Elem is pointer to Symbol_Table_Element;
First_Elem := Address(Item)
```

will cause the variable `First_Elem` to be assigned a pointer to the variable `Item`.

The relational operators (`=`, `¬=`, `<`, `<=`, `>`, `>=`) are allowed for pointers.

A pointer value is compatible with a pointer name (for assignment or comparison) only if the object types, and the ranges and attributes of the types are compatible. For example, given the declarations

```
variable V1 is pointer to (1 to 100);
variable V2 is pointer to (50 to 200)
```

assignment of `V1` to `V2` or vice-versa would not be allowed because the ranges are different. (This strict type compatibility is necessary to enforce range-checking of assignments.)

In general, pointer assignment is permitted in situations which don't allow violating range declarations or "corrupting" values.

More specifically, a `pointer to value T` cannot be assigned to a `pointer to T`, but a `pointer to T` can be assigned to a `pointer to value T`. The range restrictions on pointer assignments are relaxed slightly in the presence of `value`. For example a `pointer to (1 to 5)` can be assigned to a `pointer to value (0 to 10)` and a `pointer to character(0 to 10)` can be assigned to a `pointer to value character(0 to 100)`. The range (or length range) of the right-hand-side of the assignment must be within that of the left-hand-side. The ranges or attributes must still be such that the storage representations of the object types are the same. Thus a `pointer to (1 to 5)` cannot be assigned to a `pointer to value (0 to 10000)`, because the first numeric type uses one byte while the second uses two bytes.

The predefined constant `Null` is compatible with any pointer type. It may be used as a distinguished value to indicate the end of a linked list, etc.

The special type `unknown` may be used as the object of a pointer type.[4] The type `pointer to unknown` is compatible with any other pointer type, but values of this type may not be used to access an object. The result of dereferencing a `pointer to unknown` is an expression of type `unknown`. This cannot be assigned to or fetched. It is possible to specify its type with an open or equate statement however, or to pass it on to a procedure expecting a reference parameter.

Type `pointer to unknown` is intended for use in interfacing to external (non-Plus) subroutines for which it is not convenient or not reasonable to provide a proper type definition for all parameters. Variables of type `pointer to unknown` may also be used as a way of type-cheating, to convert one pointer type to another. (The equate statement provides a much more direct way of performing such type cheating.)

11. Record Types

A record type is used to group a series of items of other types as one conceptual unit. Each item of the record is called a field and is named with an identifier. The end of a record type is specified by `end` or `end record`.

Example:

```
type Symbol_Table_Element is
      record
         Next_Symbol is pointer to Symbol_Table_Element,
         Symbol is Symbol_Type,
         Reference_Count is Integer
      end
```

Assignment of record types is allowed. The variables assigned must be the *identical* record type (see Section K–3 in Chapter II, page 21). The operation of field selection, indicated by "`.`" is also defined. Thus given

```
variable Sym is Symbol_Table_Element
```

the field `Reference_Count` of variable `Sym` is accessed as in

```
Sym.Reference_Count := 0
```

Field selection may be applied to expressions which result in a record; for example

[4] Or, equivalently, as the type of a `reference parameter` in a procedure type description.

```
First_Elem@.Reference_Count
```

accesses field `Reference_Count` of whatever record `First_Elem` points at. If the expression results in a name, then the result of the field selection is a name; if the expression is a value the result of field selection is a value. If the expression is a constant display, the field selection will result in a constant—the value will be determined at compile time.

Note that in order to reference a field of a record, full qualification is normally required. The open statement, described in Section N, page 94, provides a way of eliminating some of the qualification.

Comparison operators `=` and `¬=` are also allowed for records, but with the same caveat as for array types: there may be padding bytes within the record layout which are not initialized and hence lead to spurious results when the records are compared.

12. Variant Fields in Records

A record type description may include a section at the end which may contain different types of items under different circumstances. Such an area is called a **variant part**. The heading of the variant part normally defines a **selector field** whose value determines how the remainder of the variant is supposed to be interpreted. It is permissible to omit the selector field.

Example:

```
type Symbol_Table_Element is
      record
         Next_Symbol is pointer to Symbol_Table_Element,
         Symbol is Symbol_Type,
         Reference_Count is Integer,
      variant Device of Device_Type from
      case Reader, Printer, Punch:
         Record_Length is Integer
      case Disk_Drive:
         Block_Size is Integer
      case Terminal:
         Rows, Columns are Integer
      end
```

`Device` is defined as a field of the record type, which is called the selector field. The value of this field is supposed to determine which of the cases that follow is in effect. The cases following may contain arbitrary lists of field definitions. The storage for any case overlays that of the other cases. The labels on the cases identify the values of the selector field for which the shared storage area should be interpreted according to the following field list. There may be more than one value specified as part of a case label.

The selector field is *not* set automatically when the variant fields are changed. Variant records provide one way of type-cheating in Plus, since it is possible to store into a shared area by referencing it with one field name and retrieve from it via another name, associated with a different type. However, the equate statement provides a much more direct way of type-cheating.

The compiler may eventually provide optional run-time facilities to check correspondence of referenced fields and the value of the selector field. The selector field may be omitted (by simply not specifying an identifier). In this case, of course, no run-time checking is possible. Note that a selector type and case labels of that type are still required in the description of a variant record. For example:

```
type Symbol_Table_Element is
      record
         Next_Symbol is pointer to Symbol_Table_Element,
         Symbol is Symbol_Type,
         Reference_Count is Integer,
      variant Device_Type from
      case Reader, Printer, Punch:
         Record_Length is Integer
      case Disk_Drive:
         Block_Size is Integer
      case Terminal:
         Rows, Columns are Integer
      end
```

is similar to the previous example, but there is no field `Device` in the record. The application program using such a variant record is assumed to know by context which variant applies.

A given record type can only contain one variant part which must always be at the end of the record. However, any field may be a nested record type which itself has a variant part.

13. Procedure Types

A procedure type defines the names and types of the parameters and result of a class of procedures.

The parameters and/or result are specified in a definition similar to a record definition. Procedures with no parameters and no result are specified as type

```
procedure
end
```

The parameters and result names and types are specified similarly to record fields, as in:

```
type Table_Search_Procedure is
      procedure
      parameter Element is Symbol,
      reference parameter Accesses is Integer
      result Position is Array_Index
      end;

/* Procedure with result but no parameters. */
procedure Getsym is
      procedure
      result Sym is Symbol
      end
```

The keyword `reference` preceding a parameter specification causes the parameters that follow to be passed by reference instead of by value. The corresponding arguments of a call of the procedure must be *names* or *constants* of the appropriate types; the addresses of the arguments are passed. A constant may only be used if the parameter type specifies the attribute `value`. The effect is exactly the same as declaring the parameters as type `pointer to ...`, using the `Address` procedure in the call to obtain the pointer to pass, and then dereferencing the parameter at each use in the called procedure.

The keyword `name` may be used instead of `reference`. This is a mostly-obsolete feature. `name` acts just like `reference` from the point of the calling routine (i.e., the address of a variable is passed). Automatic dereferencing is *not* done inside the called routine, however. That is, the parameter will appear as a pointer within the called routine.

Optional parameters may be specified following the keyword `optional`. Any optional parameters must follow all required parameters in the parameter declarations.

For example, given

```
procedure Proc2 is
      procedure
      parameter P1 is (1 to 100),
      optional parameter P2 is character(1),
         P3 is (1 to 100),
      optional reference parameter P4 is (1 to 100)
      end
```

all the following would be legal calls of this procedure:

```
Proc2(1, "A", 10, X);
Proc2(1, "A", 10);
Proc2(1, "A");
Proc2(1);
```

Procedures written in Plus may have optional parameters; however, there is currently no built-in way for them to determine the number of parameters that were passed.

System 370 Note

For compatibility with Fortran or other Type I, S-type linkage routines, if the last parameter passed is a reference (or name) parameter it will be flagged in the high-order bit. If it is not a reference-parameter, then the called routine will have to have some way of determining the number of parameters for itself.

There is currently no built-in method for a procedure written in Plus to test this high-order bit in order to determine how many parameters were passed (but see Chapter VIII, page 141).

A group of parameters that may be repeated an arbitrary number of times may be specified following the keyword `repeated`. This is only useful in interfacing to non-Plus routines, since there is currently no way of accessing the parameters from within a Plus procedure. For example, given

```
procedure Proc3 is
      procedure
      parameter P1 is (1 to 100),
      repeated parameter P2 is (1 to 100),
         P3 is character(1)
      end
```

All the following would be allowed:

```
Proc3(X);
Proc3(X, 1, "A");
Proc3(X, 1, "A", 2, "B");
...
```

But not:

```
Proc3(X, 1, "A", 2)
```

Implementation Restriction (all compilers)

Repeated parameters are not implemented yet.

Parameters may be passed in registers by using the `register` specification in the parameters declaration. Similarly, the result of a procedure may be declared to be returned in a register or several registers.[5] See Section G–5, page 75 for details of the `register` specification.

The declaration of register parameters or result only controls the way that the parameter/result passing is implemented. Within the body of a procedure, the parameters/result are not necessarily retained in registers.

The result of a procedure may be declared to be `optional`. This just means that the procedure may be used in either an expression context (requiring a result) or in a statement context (the result is to be ignored).

Example:

```
procedure Read is
   system procedure
   reference parameters
      Buffer is unknown,
      Buffer_Length is Number,
      Modifiers is MTS_Modifier_Record_Type,
      Line_Number is MTS_Line_Number_Type,
      Fdub is Fdub_Type
   optional result
     Io_Result is Dsri_Return_Types
   end
```

[5] Note that the normal System 370 Type I linkage convention (result in register 0) is *not* currently assumed, even for procedures with the `system` attribute or linkage specification. It must be explicitly stated in the type declaration.

This might be used as in

```
    Notification := Read(Buffer ... )
```

or (when the result is to be ignored):

```
    Read(Buffer ... )
```

There is also an “unspecified” procedure type. The syntax is

```
    procedure
    unknown
    end
```

An unknown procedure cannot be called, but it can be passed as a parameter or assigned to a procedure variable. The type `procedure unknown end` is compatible with any other procedure type. (It is analogous to the `pointer to unknown` type). It is intended for use in defining variables and parameters which take different types of procedure values, depending on context. Some kind of type cheating is necessary if the procedure is eventually to be called.

The type attributes `system` and `environment ...` may be applied to a procedure type description. Both affect details of the procedure call. See Section F, pages 69 and 71 for details.

Procedure type values are compatible only with other values of the identical type, with the predefined constant `Null`, or with the “unknown” procedure type. All parameterless procedures are considered compatible.

The only operations implemented for procedure types are assignment, procedure calling, and comparison. Comparison of procedure values may specify only `=` or `¬=`.

14. Global types

Global types are used only for values to be used as the “global storage environment” of a procedure.

Example:

```
    variable Psect is global("TEST")
```

The variable `Psect` holds a value which may be used to set the global storage for procedures which are defined to require `environment global("TEST")`.

The expression in parentheses following the keyword `global` is called the **global id**. It is a `bit(32)` constant, or other constant that is compatible with `bit(32)`.

The global id serves only to identify a class of compatible procedure environments. Two global types are compatible if and only if their global ids are equal. Global types are also compatible with the predefined constant `Null`.

The operations of assignment and comparison (`<`, `<=`, `=`, `¬=`, `>`, `>=`) are defined.

For details of procedure environments see Section C–3, page 46.

Implementation Restriction (PDP-11)

The type `global(...)` is not implemented for Plus-11.

F. Type Attributes

There are a number of attributes that can be applied to a type description. Attributes always precede the type description and modify its interpretation in some way.

1. Aligned

The `aligned` attribute is used to specify alignment of variables of a given type.

For the purposes of this attribute, the object machine is assumed to have a bit-addressable memory. Aligned specifies an allocation boundary requirement, an optional offset from that boundary, and whether the left or right end of the variable is to be so aligned. This attribute can only be used to strengthen the default alignment of a variable. For `bit` types, the alignment specification also overrides the default left- or right-justification of the type.

Examples:

```
type Aligned_Chars is aligned 64 left character(4);
```

Variables of this type are 4-byte character fields, aligned such that the left-hand end is at an address which is a multiple of 64 bits (i.e., doubleword aligned).

```
type Bit_24 is aligned 8 in 32 left bit(24)
```

Variables of this type are aligned such that the left-hand end is 8 bits from a fullword boundary. This is the same as specifying

```
type Bit_24 is aligned 32 right bit(24)
```

except that the first would also cause type `Bit_24` to be a left-justifying bit type, while the second would cause it to be right-justifying.

Aligned does not affect the allocated size of a variable. It just inserts or removes “filler” bytes to ensure the requested alignment.

```
type Aligned_Byte is aligned 32 right bit(8)
```

A variable of this type occupies a single byte, allocated such that the right-hand end is on a fullword boundary.

System 370 Note

For the 370 implementation, the boundary specification may be a number from 1 to 64. The offset may be a number from 0 to the specified boundary. For index types, the variable must be contained entirely within four or fewer bytes.

PDP-11 Note

Aligned is ignored by the PDP-11 compiler, except for the fields of a record. For index types, the variable must be contained entirely within a (16-bit) word.

Motorola 68000 Note

For Plus/68000, the boundary specification may be a number from 1 to 64. The offset may be a number from 0 to the specified boundary. Note, however, that many storage allocation mechanisms only give 16-bit alignment which might result in variables only being 16-bit aligned during execution.

For index types, the variable must be contained entirely within four or fewer bytes.

2. Environment

The `environment` attribute is allowed only for procedure types. It specifies the type of global storage environment that must be in effect when the procedure is called.

Example:

```
procedure Getfrom is environment Dsr_Psect_Type
   procedure
      ...
   end
```

The attribute keyword `environment` must be followed by a type description for the environment of the routine. The environment type must be one of

a. `global(`*n*`)` where *n* is a `bit(32)` or compatible constant. This indicates that the procedure uses a pseudo-register vector ("PRV") for its global storage. The constant value is required to distinguish distinct PRV environments. The default for the environment attribute is `global(%Global_Id)`.

b. `unknown`. The procedure's global environment is undefined. It may be called with any environment, and may make calls to procedures of any environment. You must ensure all such calls provide a suitable environment.

c. `pointer to` *r*, where *r* is a record type. This means that the global environment is defined by the specified record. For the System 370, to be usable as a global environment the first part of the record must have a specific format, as described in Appendix D.

Implementation Restriction (PDP-11)

`environment` isn't implemented for Plus-11.

See Section C–3, page 46 for further information about procedure environments.

3. Fast

This attribute requests the compiler to allocate variables of the type in such a way that access to them is fast if possible. This may mean using a register, or allocating in a halfword rather than a byte, etc.

Example:

```
type Subscript_Type is fast (1 to 100)
```

4. Left

The attribute `left` may be used to force a type (normally, a bit type), to be left-justifying.

Example:

```
type Four_Chars is left bit(32)
```

5. Packed

The `packed` attribute is used to request that items of the type be storage packed very closely. This generally means there will be no slack bits left except as required by alignment considerations.

Example:

```
type T1 is packed (0 to 15);
variable V1, V2 are T1
```

Without the attribute `packed`, `V1` and `V2` would each be allocated in a separate byte. With the attribute, the two will be packed into one byte.

`packed` may be specified for the fields of a record type, or the element type of an array, in order to cause the data structure to be packed.

Examples:

```
type Flags is
      record
         F1, F2, F3, F4, F5, F6, F7, F8 are packed Boolean
      end;

type Flag_Array is array (0 to 7) of packed Boolean
```

Both the above data structures use only a single byte, with each element occupying one bit.

Note that specifying `packed` for an overall record type *does not* cause the elements within it to be packed. Thus

```
type Flags is packed
      record
         F1, F2, F3, F4 are Boolean
      end
```

would occupy four bytes. The attribute in this case only affects the overall allocation of variables of type `Flags`. Since they would be byte-aligned anyway, it actually has no effect.

The object of a pointer type may specify `packed` only if the type occupies an integral number of bytes, so that all objects of the type will start at an exact byte address.

System 370 Note

For index types, a variable must be contained in four or fewer bytes.

PDP-11 Note

`packed` is ignored by the PDP-11 compiler except when applied to the fields of a record. Packed fields of a record are allocated starting from the least significant bits of each word. For index types, a variable must be contained entirely within a (16-bit) word.

Motorola 68000 Note

For index types, a variable must be contained in four or fewer bytes.

6. Right

The attribute `right` may be used to force a type (normally, a bit type), to be right-justifying.

Example:

```
type Fullword is right bit(32)
```

7. Small

`small` requests the compiler to optimize the size of the type in preference to the access time. It does not result in the extreme storage packing forced by the packed attribute. It is the inverse of the attribute `fast`. Since `small` is the default, it is never really needed.

8. System

The attribute `system` may be specified only for a procedure type. It indicates that calls to the procedure must be compatible with the standard linkage used in the operating system.

The `system` attribute affects only the code generated for procedure calls and is generally used for declaring procedures written in another language. It *does not* affect the entry/exit code generated as part of the procedure definition if the procedure is written in Plus. See the `linkage` option in Section P, page 96 for related information.

System 370 Note

For the 370 version, the `system` attribute guarantees compatibility with the OS Type I linkage.

The linkage conventions used internally in Plus are undergoing a change at the time of this edition of this document. For the older version (`%Linkage="OLD"`), the attribute `system` has no effect. With the newer version, `%Linkage="NEW"`, this attribute causes the procedure call to update a stack descriptor, so that it is later possible for the OS linkage routine to call back to another Plus linkage routine.

PDP-11 Note

The attribute `system` is ignored by Plus-11.

Motorola 68000 Note

For Plus/68000, the effect of the system attribute depends on the %Target_Operating_System compiler variable.

When %Target_Operating_System has the value "MAC/MPW" or "MAC/MDS", then the system attribute causes the compiler to generate a special instruction to call the procedure, usually an "A-line trap". The actual instruction used is given by the external name of the procedure, which must be an even number of bytes (characters) in length, usually specified as a hexadecimal constant. Note that this implies that there cannot be variables of a system procedure type, as there is no implemented way to call them.

When %Target_Operating_System has the value "AMIGA", then the system attribute causes the compiler to call the procedure via an offset from the global base register (which may first be loaded by any with phrase in the procedure call). This is used to call "system library" routines.

For example with the Macintosh

```
procedure Set_Port is system
      procedure
      parameter Gp is Graf_Ptr
      end external 'A873'
```

defines the Set_Port routine to be the A-line trap A873. For the AMIGA,

```
procedure Open_Window_Procedure is
      environment pointer to Intuition_Base_Type
      system procedure
      reference parameter New_Window is value New_Window_Type
            in register A0,
      result Window is pointer to Window_Type
            in register D0,
      end external "_LVOOpenWindow"
```

defines the window opening procedure for the "Intuition" library. When this procedure is called, the call will be made relative to the Intuition_Base_Type that is supplied on the call.

See Appendices D, E and F for further information about Plus linkage conventions.

9. Value

The value attribute specifies that names of the type are to be automatically denamed into values whenever referenced in the program. Thus, the value attribute prevents assignment. It is mainly used as an attribute of the object type for a pointer, in order to allow the pointer to point to a constant.

For example, given

```
variable P is pointer to value character(10)
```

the assignment

```
P@ := "abcdefghij"
```

would be invalid, since dereferencing `P` produces a value, which cannot be assigned to.

The compiler knows that the object pointed at by `P` cannot be changed, so `P` is allowed to point to a constant:

```
P := Address("ABCDEFGHIJ")
```

would be allowed. The `Address` function produces a pointer to a value when its argument is a constant. This can only be assigned to a type with the `value` attribute. The pointer variable with the `value` object may, however, have a pointer to a name assigned to it.

The attribute `value` may also be useful when an external variable is to be treated as "read-only" within the program. For example:

```
variable Caseconv is value character(256) external
```

guarantees that the compiler will issue an error message if the program contains any statement that might attempt to store in `Caseconv` (either directly or indirectly via a pointer).

G. Declarations

The constant, variable and type declarations have similar syntax. Examples of declarations are:

```
constant Max_Sym_Length is 10;

type Symbol is character(0 to Max_Sym_Length);

variable Msg is pointer to Stream_Type;

procedure Getsym is
      procedure
      result Sym is Symbol
      end
```

A list of identifiers may appear where the single identifier is declared in each of the above (on the left of the keyword `is`). It is also permissible to combine a series of declarations with a single use of the appropriate keyword. Thus:

```
variables Low, High, Pos are Array_Index,
   Str is Varying_String
```

1. Constant Declarations

The constant declaration is used to associate an identifier with a constant value of any type. The value may be expressed as a constant or as an expression all of whose terms are constants. Once a constant-identifier has been declared it may be used in any situation requiring a constant.

Examples:

```
constant Max_Sym_Length is 10;
constant Max_Number_Symbols is 600
```

2. Entry Specification

A constant declaration may specify the keyword `entry` following the constant expression. This causes the constant to be generated as a separate csect in the object module.

Example:

```
type Procedure_Vector_Type is array (Open#, Do_It, Close) of
      procedure
      end;
constant Procedure_Vector is Procedure_Vector_Type(Open_Procedure,
      Do_It_Procedure, Close_Procedure) entry "PROCVECT";
```

will produce a csect containing the addresses of the three procedures.

The `entry` keyword may be followed by a string specifying the external symbol to use—`PROCVECT` in the above. If the external name is not given, it will be generated from the constant identifier by taking the first four letters and the last four letters.

Entry-constant declarations are most often used with structure (array and record) constants, so that each routine referencing the constant doesn't have its own copy of the constant.

For simple constants, it is possible that a routine may have its own copy of the constant value, even if an entry constant declaration is used to define it. This is because the compiler uses various techniques in accessing constants, some of which do not actually require a value in the literal pool of the program.

Implementation Restriction (Motorola 68000)

When `%Target_Operating_System` has the value `"MAC/MDS"`, Plus/68000 does not implement `entry` constants, due to limitations in the MDS linker.

3. Type Declarations

The type declaration is used to associate an identifier with a type description. Many examples have already been given.

It is not necessary to associate an identifier with *every* type by means of a type declaration. It is perfectly permissible to use the type description directly within a variable declaration. However, be warned that Plus does absolutely *no* equivalence calculations for record, array or procedure types in determining type compatibility. For these types, a single type description is required, either by using a type declaration, or by declaring all relevant variables in the same variable declaration. See Section K–3 in Chapter II, page 21.

4. Variable Declarations

The variable declaration is used to allocate storage for a variable of a specified type. Allocation for local variables is normally done on a stack, which is pushed and popped at procedure entry and exit only. The stack-top at different points within a procedure (i.e., as scopes are begun and ended) is determined at compile time. Global variables are allocated in global storage at program load time.

It should perhaps be noted, for those familiar with Algol-W records, that records in Plus are treated no differently from any other type. They are not dynamically created by references to them.

An external dynamic-allocation mechanism is easily implemented within the language by defining a routine to return a pointer to a record of the required type. This routine could then allocate a record by calling the MTS `Getspace` routine.

5. Allocation Specifications

There are several additional specifications that may appear in the variable declaration, *following* the type. They affect the way the variable is allocated or accessed.

a. External Allocation

The `external` phrase may be used in a variable declaration to specify that the variable is allocated (at load time), externally to the Plus program. This is typically used to access tables defined by other programs. Plus `entry` constants may also be used to define such tables.

The external symbol to be used may also be specified as a string constant following the keyword `external`. If the symbol is not specified, one is generated from the variable name, as described in Section C–6, page 49.

Example:

```
variable Caseconv is character(256) external;
variable Parsetab is Syntax_Tables_Type external;
variable Ascii_To_Ebcdic is character(256) external "ASCEBC"
```

b. Register Allocation

The declaration of a variable may specify that the variable is to be allocated in a general register or a range of contiguous registers.

This specification may also be applied to the declaration of procedure parameters and results in a procedure type description. When it is used for parameters or results, the specification affects only the way that the data is passed between the caller and called procedure. It does not necessarily cause the variable to remain in a register inside the called procedure.

Examples:

```
variable Temp is Integer in register; /* any register may be used. */
variable Temp2 is Integer in register 2;

procedure Freespac is
      system procedure
      parameter Flag is Fullword in register 0
      reference parameter Location is unknown in register 1
      end;
```

```
procedure Julgrgtm is
      system procedure
      parameter Jultim is Integer is register 1
      result Grgtim is character(16) in registers 0 to 3
      end
```

The compiler may reserve certain registers for its own use and not allow them to be used for register variables. Each compiler provides certain predefined register variables that allow access to any reserved registers that may have to be manipulated for special linkage applications. See Section R, page 100 for details. An error message will be issued if you attempt to use a register that is in use by the compiler. The register specification may be used for efficiency reasons, to assist the compiler in code-generation. However, we expect the compiler to do a reasonable job of register allocation (eventually).

The register specification should also be used in conjunction with the `Inline` procedure, to specify variables for the registers required when generating machine-code.

A variable that is allocated in a register cannot be passed by reference to another procedure, nor can it be used with the `Address` built-in procedure.

A parameter or result that specifies register allocation *may* be passed to another procedure by reference or used with `Address` under some circumstances. The register specification forces the parameter to be allocated as the size of the register (32 bits for Plus/370 and Plus/68000, 16 bits for Plus-11). If this is different from the normal size of the type, it will not be possible to pass it by reference, because the called procedure would not correctly access the storage area. A compiler error message will be issued.

Implementation Restriction (System 370)

Currently, `register` may not be specified for array, record, character, real or left-justifying bit type variables, but may be used for parameters and results of any type provided the appropriate number of registers are specified.

If the register attribute is used for a parameter of type `real(`*n*`)`, a *general* register, not a floating point register will be used.

A range of registers may be used for parameters and results, but not for variables.

A procedure which returns more than one result in registers may be defined by first defining a record-type corresponding to the set of values returned, then declaring the procedure to return this record type in the appropriate register range.

Implementation Restriction (PDP-11)

`register` may not be specified for array, record, character, or left-justifying bit type variables, parameters or results.

A range of registers may not be specified for the register allocation.

Implementation Restriction (Motorola 68000)

`register` may not be specified for array, record, character, or left-justifying bit type variables, parameters or results.

A range of registers may not be specified for the register allocation.

The register numbers 0 through 7 indicate registers D0 through D7, with 8 through 15 used for A0 through A7.

c. Absolute Allocation

The `absolute` phrase may be used in a variable declaration to specify that the variable is located at a specified *machine address.* This, of course, is mainly useful in generating highly machine-oriented code. For example:

Examples:

```
variable Svc_Old_Psw is Psw_Type at absolute '20';
variable Memory is array bit(24) of bit(8) at absolute 0
```

The latter declaration allows any byte of memory to be accessed using its address as an index.

6. Procedure Declarations

The procedure declaration is used to define an identifier or list of identifiers to be procedure constants. The procedure declaration normally specifies the type of the procedure, which in turn determines the identifiers and types of the parameters and result. The type may be omitted from a procedure declaration, in which case the simple type `procedure end` is assumed.

A procedure identifier must be declared before the procedure can be defined, called, or assigned to a procedure variable. Procedure declarations obey the same scope rules as other declarations. Thus a declaration given inside a procedure or nested scope is forgotten at the end of that scope while an external declaration remains in effect for the remainder of the compilation.

A procedure declaration may be combined with the procedure definition. In this case, the procedure declaration is considered external to the procedure.

Examples:

```
procedures Read, Write are Io_Parameter_Type;
procedure Parameterless_Procedure
```

The latter is equivalent to

```
procedure Parameterless_Procedure is
      procedure
      end
```

Note that if a series of procedure declarations are connected together (as with variable or type declarations), the type may be omitted only from the last list. That is, the declaration

```
procedures Read, Write is Io_Parameter_Type,
   Parameterless_Procedure
```

is equivalent to

```
procedure Read is Io_Parameter_Type;
procedure Write is Io_Parameter_Type;
procedure Parameterless_Procedure;
```

7. Procedure Specifications

There are several additional specifications that may appear in a procedure declaration, following the procedure type. When more than one is used, they may appear in any order.

a. External

A procedure declaration may specify an external symbol to be used instead of the procedure identifier.

Examples:

```
procedure Get_From_User is Io_Parameter_Type external "GUSER";

procedure Get_User_Info is
      procedure
         ...
      end external "GUINFO"
```

b. Linkage

The `linkage` specification is used to request a special entry/exit sequence. It is generally given as part of the heading for a procedure *definition*, but may appear in the procedure declaration instead. (For declarations that are to be included from libraries, it is sometimes more convenient to attach the linkage specification to the declaration.)

The allowed options are described in Section P, page 96.

Implementation Restriction (Motorola 68000)

`linkage` is not implemented in Plus/68000.

Example:

```
procedure Main is Main_Procedure_Type linkage "PLUSENTR"
```

c. Stacksize

The `stacksize` specification indicates the size of the run-time stack that should be provided when the procedure is called.

This option is currently used only as part of the entry/exit code of a procedure. It is ignored by the caller.

The stacksize specification is mainly used by procedures that have special entry/exit code to initialize the Plus run-time setup. The value is made available to the entry code, which can use it in allocating a stack.

For Plus/370, if the stacksize specification is given and the compiler option `%Stack_Check` is True, the code generated will check the amount of stack available against the value of `stacksize`, rather than using the actual requirements of the procedure.

See Appendices D, E and F for details of the entry/exit code and stack setup required.

Example:

```
procedure Special is Main_Procedure_Type stacksize 4096
```

H. Constants

A Plus program may contain constants of various types. For each scalar type, the language defines a denotation for values of that type. For structured types (arrays and records), constants are constructed by using a type name, and a list of values for the components of the structure, as described below.

Certain "constants" have values which are determined at the time the program is loaded, and hence are unknown at compile time. Such constants are not valid in contexts which require knowing the value at compile-time, such as array dimensions, constant expressions, etc.

1. Integer constants

Integer constants have the normal decimal representation. The range of values depends on the object machine; it will always include all integers which the object machine supports as the basic instruction level.

2. Character constants

Character constants are enclosed in the character quote ("). A quote within a constant is represented by two quotes.

The character set is machine-dependent. For the System 370 version, EBCDIC is assumed. For the PDP-11 and Motorola 68000 version, character constants are translated to ASCII.

3. Bit constants

Bit constants are denoted by enclosing a series of digits in apostrophes ('). By default the digits are considered to be hexadecimal, but a different base may be specified.

Examples:

```
constant S8_Punch is 'E0';
constant Bit_Example is '(1)10 (3)707'
```

A base is specified by giving a power-of-two radix in parentheses as part of the bit string. Thus the second example denotes a bit string of 11 bits, consisting of `10` in binary (base 2|1) followed by `707` in octal (base 2|3); i.e, the binary value is `10111000111`.

4. Real constants

Implementation Restriction (PDP-11)

Real constants are not implemented for Plus-11.

Implementation Restriction (Motorola 68000)

Real constants are not implemented for Plus/68000.

Real constants have the same syntax as in Fortran and many other languages, i.e., a decimal integer and/or fraction followed by an optional signed exponent. Up to 34 significant digits are allowed by the 370 implementation. The following are all legal examples:

```
  1.0
 10.579
   .5
  2E-60
  3.14E20
```

Exponents are always indicated with `E` (or `e`)—the “`D`” and “`Q`” forms used in Fortran are not used.

All real constants in Plus programs are converted to extended precision (16 byte) forms. A constant can be explicitly coerced to a shorter length by using a constant display as in the following example:

```
type Short_Real is real(7);
   ...
constant Pi is 3.1415926535879,  /* extended precision */
   Short_Pi is Short_Real(Pi)    /* single precision */
```

A real constant will be *rounded* when it is converted to a shorter length.

Implementation Restriction (System 370)

It is intended that constants should be automatically coerced to shorter forms as required by context, but this is not implemented yet. Explicit conversion as described above must be used.

5. Constants of identifier-list types

The names of the elements of the type form the constants of that type. See Section E–4, page 57.

6. Procedure constants

The name of a procedure is a procedure constant of the specified type. A procedure constant is always a "load-time" constant; i.e., the value is not known at compile-time.

7. Pointer constants

Under certain circumstances, the result of the `Address(...)` function will be a constant. Currently, this will happen if and only if the argument of `Address` is a constant or an external variable. Pointer constants are always load-time constants; i.e., the value is unknown at compile time.

Implementation Restriction (Motorola 68000)

When `%Target_Operating_System` is set to `"MAC/MDS"`, then Plus/68000 does not implement pointer constants, due to limitations in the MDS linker.

8. Constant Displays

A constant display is a type name followed by a parenthesized list of constants. It is used to create a constant of the given type. This is most often used for the creation of structure (array and record) constants. A constant display may be used with scalar types to control the storage representation of the value (see examples below).

For array types, the constants in the list must be suitable for the elements of the array. The number of elements must agree with the bounds of the array. For record types, the constants must be suitable for the fields of the record. If the record has variants, a constant must be specified for the selector tag, even if the selector is not defined as a field. The constants which follow the tag are then the ones required for that variant, if any. With any other type, there will be only one element in the list, and it must be a constant whose type is compatible with the type-name of the display.

Examples:

```
type Awry is array (1 to 5) of character(1),
      Rec1 is
         record
            F1 is character(2),
         variant (Red, Green, Blue) from
         case Red:
            F2 is character(0 to 5)
         else
            F3 is (-32768 to 32767)
         end,
      Rec2 is
         record
            F1 is character(1),
         variant F2 is (0 to 10) from
         case 1:
            F3 is fast bit(6)
         end,
            L_Bit_32 is aligned 32 left bit(32),
            Short_Real is real(7)
```

Given the above definitions, the following are valid constant displays:

```
    Awry("a", "B", "c", "d", "e");
    Awry('00', "Z", '40', " ", "0")
```

Note that all five elements must be given.

```
    Rec1("ab", Red, "abc");
    Rec1('00', Green, 5)
```

Note that the selector field for the variant is given, although it does not appear in the actual constant. The selector field determines which case of the variant is used to interpret the constants that follow it.

```
    Rec2("a", 0);
    Rec2("B", 1, '0')
```

In this case, the selector field forms an element of the constant. When it has the value 0, there are no other fields in the record.

```
    L_Bit_32("ABCD");
    L_Bit_32('0001')
```

This example shows the use of a constant display to force the specified constants to be treated as fullword-aligned, left-justified, `bit(32)`. Without the display, the first constant (`"ABCD"`) would be a `character(4)` (byte aligned), and the second would be a right-justified `bit(32)`.

```
    Short_Real(10.5)
```

This example shows the use of a constant display to force `10.5` to be single precision.

Constant displays can be used in any context in which a simple constant is allowed. For example, a table might be defined in Plus using a constant declaration such as

```
    constant Special_Characters is Awry("+", "-", "*", "/", """")
```

which could then be used as appropriate in the program:

```
    do I := 1 to 5
       return when Char = Special_Characters(I)
    end
```

etc.

As another example, a record might be initialized using a record constant:

```
    variable V is Rec1;
       ...
    V := Rec1("ab", Red, "abc")
```

Note

A constant display will be allocated as a separate object module only if it is declared in a constant declaration with the `entry` attribute. Otherwise it may be emitted as part of the "constant pool" for *each* procedure that references it.

Implementation Restriction (System 370)

Constants which should be doubleword-aligned (i.e, the type is `aligned...64`) will get the specified alignment only if they are within a record or array.

9. Constant storage representation

A constant may appear as a parameter of `Address(...)`, or as a reference-parameter (in some situations).

In order that the resulting pointer object can be processed consistently, and type- and range-checked where necessary, all non-structure constants have associated with them a default storage representation.[6] This, in effect, provides a more specific type-definition for the pointer created.

The default storage representation can be changed by using a constant display.

The default representations used are:

integer	fullword (32 bits on 370 and 68000, 16 on PDP-11).
character	`character(`n`)`, where n is the length of the string.
bitstring	if the length is less than the word size, then fullword, otherwise the allocation is `character(`byte-length`)`.
pointer	fullword
procedure	fullword
id-list type	fullword
real	extended precision (16 bytes)

I. Expressions

Expressions in Plus are formed in the usual way, by combining various operands with appropriate operators and parentheses.

1. Operands and Operations

The primitive operands out of which an expression is composed include constants, symbolic constants, variable names and procedure names. The repertoire of operations includes all the usual arithmetic and logical operators, subscripting array names, selecting fields of records, following pointers, and calling procedures with an appropriate list of parameters.

6 The storage representation of a structure constant is determined by the type of the constant.

The language strictly controls which operators may be applied to different types of operands. Certain operators can be applied to various types of operands, but the semantics may depend on the types of the operands. For example if `V1` and `V2` are numeric-type variables, then `V1 < V2` denotes the arithmetic comparison of their values, while if `V1` and `V2` are character types, it denotes a logical comparison.

Plus expressions follow normal precedence rules for the arithmetic operators. Rather than introducing a complex precedence hierarchy, most other operators are given precedence equal to the arithmetics. The complete precedence hierarchy is as follows:

1 (highest)	unary operators `+`, `-`, `¬`, `not`, `abs`
2	multiplying operators `*`, `/`, `mod`, `&`
3	adding operators `+`, `-`, `||`, `|`, `xor`
4	relational operators `<`, `<=`, `>`, `>=`, `=`, `in`, `subset`, and negation of each. Negations may be specified with `¬` or `not`— e.g., `not=`, `¬=`, `not<=`, `¬<=`, etc.
5	`and`
6 (lowest)	`or`

2. Coercions

Plus will perform a certain set of operations automatically if the operations are required by the context in order to make operands of other operators type-compatible. In Algol-68 terminology, such operations are **coercions**. The coercions that Plus will perform include fetching the value of a storage location (denaming), converting a value into a singleton set containing that value, conversion in either direction between bit-types and other scalar types. Details of the operations and coercions applicable to different types, together with examples, are given in the sections describing the types.

3. Logical Expressions

A **logical expression** is really just an expression whose value may be zero (false) or one (true). Any numeric value may be used as a logical expression, however, with any non-zero value treated as true. The main use of logical expressions is in if statements, although they may be used in other contexts. The comparison operators (`<`, `>`, `=`, `¬=`, etc.) result in logical expressions. Logical expressions can be combined by using the special logical operators `and` and `or`. These operators are sometimes called the "McCarthy-and" and "McCarthy-or". The semantics are such that the second operand is *not evaluated* if the outcome of the logical expression can be determined from the first. Thus,

```
if I <= Max_Number_Symbols and Table(I) ¬= Test_Elem
then
   ...
end if
```

is executed as if it were:

```
if I <= Max_Number_Symbols
then
```

```
      if Table(I) ¬= Test_Elem
      then
         ...
      end if
   end if
```

Arbitrarily complex logical expressions can be built up out of `and` and `or`; however, to avoid misunderstandings, the language requires that if the two operators are combined in an expression, parentheses *must* be used to indicate the intended order of evaluation.

The logical operator `not` is defined to give a result of zero or one always. It is zero if and only if its operand is non-zero.

Warning

The operator ¬ is a *bit-string* operator whose result is the *complement* of its operand. This does not give the same result as the logical operator `not`.

```
¬false = ¬0 = 'FFFFFFFF' = -1 = true
¬true = ¬1 = 'FFFFFFFE' = -2 = true
```

Thus both act as `True` if used in a logical expression.

J. Assignment Statements

Simple assignment is denoted by `:=` in Plus. Multiple assignments may be specified by separating left-hand-sides by commas. The right-hand-side of a multiple assignment is evaluated once only.

Example:

```
Low, High := 0
```

You can also specify an operator in conjunction with assignment. The statement

```
Table_Size +:= 1
```

is a shorthand for

```
Table_Size := Table_Size + 1
```

Similar notation can be used for any of the binary operators `+`, `-`, `*`, `/`, `mod`, `||`, `|`, `&` or `xor`, and for any left-hand-side expression.

Warning

Certain assignments (mainly of character types) may build the result directly in the left-hand-side variable. Thus the expression forming the right-hand-side should not depend on the previous value of the left-hand-side. For example,

```
Var1 := Var2 || Var1
```

will first move the value of `Var2` into `Var1`, and therefore produce the wrong result when `Var1` is concatenated onto it.

At the moment, this situation is not usually detected by the compiler.

K. Procedure Calls

Procedures may be called as self-contained statements, or as elements of expressions in other statements. Procedures with no return value are called by simply specifying the procedure identifier, with parameter list (possibly null), as a separate statement, as in Algol. Procedures that are declared to return a value are called in an expression in the usual way, with parameters, if any, given as a parenthesized list following the procedure identifier (or expression resulting in a procedure value). In both cases, if the procedure has no parameters, an empty parenthesis pair `()` `must` appear after the identifier.

Examples:

```
Elem := Getsym();

Message(Msg, "Error - too many symbols.</>");

Pos := Linearsearch(Test_Elem, Accesses)
```

1. Parameter Passing

Parameters may be passed by *copying* the value (this is traditionally known as "call-by-value"), or by passing a pointer to the argument (known as "call-by-reference"). The type of the procedure's type description specifies which kind of parameter passing is required for each parameter.

The default is call-by-value. This applies to any parameter type, including arrays and records. In general, if an array or record is to be used as a parameter, it is preferable to pass a pointer to it, either explicitly (by declaring the parameter type as `pointer to ...` and passing `Address(...)`), or implicitly by using a reference parameter.

When call-by-reference is used, there are restrictions on the possible arguments that may be used. A reference argument must be a name (or in some cases a constant). Expressions, except those resulting in a name, are not possible. In addition to being assignment-compatible with the type of the parameter, the type of the argument must obey the stronger requirements of pointer compatibility, as described in Section E–10, page 61.

A constant may be passed by reference only if the parameter type has the attribute `value`, which guarantees that the pointer object cannot be stored into from the called routine.

When a constant is passed by reference, the compiler will perform a coercion on the constant to the form required by the parameter, if possible, before obtaining the address. This coercion is equivalent to implicitly using a constant display to set the constant type.

For example, given

```
procedure Test is
      procedure
      reference parameters Str is value character(1 to 10),
         Num is value fast (0 to 10)
      end
```

then the call

```
Test("abcd", 9)
```

will result in the passing of a pointer to a varying string consisting of the length 4 (in a byte) and the characters `"abcd"` for the parameter `Str`, and a pointer to a halfword (System 370) constant `9` for the parameter `Num`. (Without the implicit coercion, the constant `"abcd"` would be of type `character(4)`, and `9` would be a fullword integer. The pointers to these would then be incompatible with the reference-parameters.)

2. Return Codes

For compatibility with procedures written in other languages, which may return a "return code", the value of the return code may be obtained by specifying a `return code` variable as part of the call. For example:

```
variable Rc is Number;
Scards(Buffer, Buflen, Mods, Lnum, return code Rc);
if Rc ¬= 0
then ...
end if
```

The expression following the phrase `return code` must be a name expression of any index type.

There is currently no built-in method for a procedure written in Plus to set a return code to return to its caller. Generally, function results or reference parameters are used to return information to the caller.[7]

System 370 Note

The System 370 return code is assumed to be returned in general register 15.

Implementation Restriction (PDP-11)

`return code` is not implemented in the PDP-11 version.

Motorola 68000 Note

Plus/68000 assumes the return code is in register `D0`.

3. Switching Global Storage Environment

If the procedure being called requires a different global storage environment from the caller, the procedure call *must* provide the address of the required global storage. This is done by using the `with` phrase in the procedure call.

Given the following declarations:

[7] However, see Chapter VIII, page 144.

Examples:

```
procedure P1 is environment global("QQSV")
      procedure
      ...
      end,
   P2 is environment global("FOO")
      procedure
      ...
      end,
   P3 is environment pointer to Rec_Type
      procedure
      ...
      end;

variable V1 is pointer to Rec_Type,
   V2 is global("QQSV"),
   V3 is global("FOO")
```

the following would be legal calls from any environment:

```
P1( ..., with V2);
P2( ..., with V3);
P3( ..., with V1)
```

If `return code` and `with` are both used, they may occur in either order.

Implementation Restriction (PDP-11)

Plus-11 does not support the mechanisms for switching global storage environments.

L. Control Structures

Plus includes control structures for selecting between alternatives (the if statement and the select statement), for looping (the cycle and do statements), and for exiting and repeating a block of statements. There is *no* goto statement.

1. If Statements

The if statement in Plus is a bracketed construct, terminated by `end` (or `end if`). The then-part and else-part are each a scope block. That is, a sequence of declarations and executable statements may appear as the body, without requiring the use of `begin...end`.

The else-part is optional.

Example:

```
if Element < Table(Pos)
then
   High := Pos - 1
else
   Low := Pos + 1
end if
```

Nested if statements may be abbreviated using the `elseif` clause. The statement

```
if Return_Code ¬= 0
then
   Sym := ""
else
   if Length(Str) > Max_Sym_Length
   then
      Message(Msg, "Error - symbol too long</>");
      Sym := Substring(Str, 0, Max_Sym_Length)
   else
      Sym := Str
   end if
end if
```

may be replaced by

```
if Return_Code ¬= 0
then
   Sym := ""
elseif Length(Str) > Max_Sym_Length
then
   Message(Msg, "Error - symbol too long</>");
   Sym := Substring(Str, 0, Max_Sym_Length)
else
   Sym := Str
end if
```

This process may, of course, be repeated for further nested if's. There is only one `end if` to terminate an arbitrary `if... elseif... elseif...` sequence.

Implementation Restriction (all compilers)

Currently, a compiler "parse stack overflow" will occur if an if statement contains a sequence of more than about 25 `elseif`'s.

2. Select Statements

The select statement allows a multiple-way branch according to the value of a given expression. In effect, it is a generalization of the if statement to types other than Boolean. (This statement is similar to what is called a case statement in Pascal.)

The heading of the select statement specifies an expression whose value determines the case to be executed. Note that the range is not restricted to numeric types; any "index type" (see Section E–6, page 58) is allowed.

The body of a select statement is a series of cases. Each case consists of a scope block, preceded by a label specifying one or more constants which are the values of the selection expression for which this case is to be performed. After completion of execution of the statements in the selected case, execution continues following the select statement. The end of the select statement is delimited by `end` or `end select`.

Note that a given constant may be used as a label on at most one case. A list of values may be given for the label on a case. The select statement may specify an `else` case which is to be executed for any values that have not been specified.

Example:

```
select Device from
case Reader, Punch:
   Record_Length := 80
case Printer:
   Record_Length := 132
case Terminal:
   Rows := 25;
   Cols := 80
else
   Snark()
end select
```

Select statements are currently implemented in all compilers by using branch tables. This provides for fast execution, but the branch table may get quite large. The branch table will contain one entry for *every* value between the lowest and highest case labels used. So, for example,

```
select I from
case 1: ...
case 1000: ...
end select
```

will generate a branch table with 1000 entries (two bytes each), even though there are only two actual cases specified.

Eventually there may be a "skip-chain" implementation of sparse select statements. At present, it may be preferable to use `if... then... elseif... then... end if` in some situations.

3. Cycle Statements

The cycle statement is a general looping construct. The body of the cycle (which is again a scope), is executed repeatedly until terminated by execution of either a return or exit statement, described below. The end of the cycle statement is marked by `end` or `end cycle`.

Example:

```
cycle
   variable Elem is Symbol;
   Elem := Getsym();
   exit when Length(Elem) = 0 or Elem = "/end";
   if Table_Size >= Max_Number_Symbols
   then
      Message(Msg, "Error - too many symbols.</>");
      exit
      end if;
      Table_Size +:= 1;
      Table(Table_Size) := Elem
   end cycle
```

4. Do Statements

Plus contains two limited forms of do statements.

One form allows for looping with an increasing or decreasing index. It is restricted to an increment or decrement of one only. An increasing loop is indicated by the keyword `to`, while a decreasing loop is indicated by `downto`.

Example:

```
do Pos := 1 to Table_Size
   Accesses +:= 1;
   return when Table(Pos) = Element with Pos
end
```

The second form of do loop is intended ultimately to allow stepping through the members of a specified set value. Currently, this is implemented only for a special case in which a type-identifier is given to specify the set of values to be stepped through.

Example:

```
type Device_Type is (Printer, Reader, Punch, Tape_Drive, Disk_Drive,
   Terminal);
variable D is Device_Type;
do D := each Device_Type
    ...
end
```

The order in which the `do...each` form steps through the set is up to the compiler—if a particular order is required you should use `do...to` or `do...downto` to specify it.

The limits of the loop are determined at the time execution of the loop begins; modification of the final value of a `do...to` or `do...downto` loop within the loop will have no effect. It is possible (though generally not good practice) to modify the control variable within the loop.

The body of a do loop is also a scope block. The exit statement can be used to terminate a do loop before its limit is reached. The end of a do loop is indicated by either `end` or `end do`.

The value of the control variable upon termination of a do loop is always the value that it had during the last execution of the loop. In the case of a loop that executes zero times, the value of the control variable will not be changed from the value it had before execution of the do loop heading.

Implementation Restriction (all compilers)

The control variable of a do loop must currently be a simple variable identifier—array elements and other name expressions are not allowed.

5. Begin Blocks

The begin block consists simply of a scope block surrounded by `begin...end`. It is mainly used for one of two reasons:

a. To restrict the scope of local variables, open statements and equate statements to the series of statements for which they are required.

b. To delimit a series of statements from which it is desired to escape with the exit or repeat statements.

Example:

```
begin
   variable Temp_Fdub is Fdub_Type;
   /* Exchange new and old Fdubs */
   Temp_Fdub := New_Fdub@;
   New_Fdub@ := Old_Fdub@;
   Old_Fdub@ := Temp_Fdub
end
```

6. Compounds

The term **compound** refers to cycle statements, do loops, or begin blocks. The statements `exit` and `repeat` may be used inside a compound to branch to the end or beginning (respectively) of that compound.

A compound may be labelled by preceding it with an identifier surrounded by `<` and `>` (such as `<Outer>` in the example below), and following it with the same identifier. This label may be used in exit and repeat statements inside the compound to refer to it. Normally, the exit or repeat statements refer to the closest enclosing compound. Compound labels allow exiting more than one level.

Note that *only* the specified statements form compounds; `exit` and `repeat` cannot be used to branch out of if statements or select statements, unless the statements are embedded in `begin...end` or another compound.

With labelled compounds and multi-level exits, it is possible to synthesize complex control flows that in most languages cause one to resort to the use of goto statements. However, it should be noted that if they are used indiscriminately, it is possible to produce programs that are just as entangled as if gotos were used.

7. Exit Statements

The exit statement is used to branch out of a compound. The statement may specify a condition under which the exit is to be taken. That is,

```
exit when Something
```

is equivalent to

```
if Something
then
   exit
end;
```

Exit conditions may also be specified in the form `exit unless....`

`exit` normally leaves the closest enclosing compound. More than one level can be escaped by labelling the compound to be exited and specifying the label in the exit statement.

Example:

```
<Outer>
cycle
   ...
   do I := 1 to Max_Symbols
     ...
     exit <Outer> when Elem = Table(I);
     ...
   end;
   /* inner loop completed normally */
   ...
end <Outer>
```

In this example, the exit statement inside the do loop will exit both the do loop and the cycle containing it.

8. Repeat Statements

The repeat statement is similar to the exit statement, except that instead of *terminating* the loop, it branches back to the beginning of the compound, and resumes execution of the compound from that point.

When used in a do loop, repeat "steps" to the next iteration of the loop. That is, the control variable will be incremented or decremented and compared to the limit to decide whether to re-enter the body of the loop or terminate the loop.

Conditions and labels are allowed in repeat statements as for exit statements.

Example:

```
cycle
   Scards_Varying(Str, Rc);
   exit when Rc ¬= 0;
   repeat when Length(Str) = 0;
   /* Process the input record. */
   ...
end
```

In this example, the loop terminates via the exit when `Scards_Varying` returns a non-zero return code, but returns to the beginning of the loop and repeats the read if a null line is read.

9. Return Statements

The return statement is used to return from the procedure containing it. A return statement may specify a condition, as in exit and repeat statements. It may also specify a return value, if the procedure's type specifies a return value.

If a procedure is to return a value, its type description specifies an identifier whose value is returned by default when the procedure returns. If the return statement does not specify a return value, then the value of this identifier is used.

If a condition (`when` or `unless`) and a return value (`with`) are both specified, they may appear in either order.

A return is automatically performed at the end of a procedure.

Examples:

```
return when Table(Pos) = Element with Pos;
return with 0
```

M. Assert

The assert statement can be used to incorporate special run-time checks into a program for debugging purposes. Code is generated for an assert statement only if the compiler option `%Assertion_Check` has the value true. If this option is false, the assert statement is treated as a comment.

The assert statement specifies a logical expression which is to be evaluated when the program is executed. If the expression is true, execution continues normally. If it is false, execution of the program is terminated (in MTS, in a `RESTART`able way) with an error message.

When coding an assert statement, you should take care that there are no side effects of the statement that might cause the operation of the program to change if assertion checking is later disabled. That is, the expression in the statement should not change the value of any variables in the program or otherwise modify its operation.

Example:

```
assert P1 ¬= Null;
P1@ := 0
```

If `P1` is `Null` when the assertion is executed, execution will be terminated with an error message.

N. Open Statements

The open statement allows accessing fields of a record without the necessity of specifying the record name. (It is similar to the "with record" prefix of Pascal.)

The open statement is treated like a declarative statement, but may only occur within a procedure body—it is not allowed in a global block or external to a procedure. It is in effect from the point at which it occurs, for the remainder of the scope block. The effect of this statement is to "redeclare" the fields of the specified record as if they were separate variables, for the duration of the scope.

The open statement specifies a name expression. It is possible to open elements of arrays of records or pointers to records.

Example:

```
type Symbol_Table_Element is
      record
         Symbol is Symbol_Type,
         Reference_Count is Integer
      end;
variable Symbol_Table is array (1 to Max_Number_Symbols) of Symbol_Table_Element;
...
open Symbol_Table(I);
...
Reference_Count +:= 1
```

Following the open statement, for the remainder of the scope block containing it, the identifier `Reference_Count` refers to that field of element `Symbol_Table(I)`.

Apart from notational convenience, this can be an efficiency consideration, since addressability to the required record will be obtained at the time the open is performed.

It should be noted that the name expression in an open statement is evaluated at the time the open is performed. Subsequent changes to pointers or array subscripts involved will have no effect on the locations accessed when the field names are used.

The open statement may also specify a record-type, which redefines the type of the name being opened. For example,

```
open Ptr@ as Symbol_Table_Element
```

specifies that, regardless of the actual type of `Ptr@`, it is to be treated as if it were `Symbol_Table_Element` for the purposes of the open. This provides another form of type cheating in the language. It is most often useful when processing records pointed at by pointers of type `pointer to unknown`, in order to specify the object type of the pointer.

O. Equate Statements

The equate statement provides a way of associating a new identifier with an existing storage location, and optionally associating a different type with the location. The statement in effect declares a new variable, and specifies that it is to be overlayed on a storage area defined previously. The identifier declared remains defined for the remainder of the scope in which the statement occurs.

Equate provides an "official" way of type-cheating in Plus. It is also sometimes convenient to use an equate definition to avoid repeating long name expressions. Equate also contributes to program efficiency, since addressability to the specified locations is obtained (if necessary) at the time the equate statement is performed.

Judicious use of this facility *can* improve program clarity, by removing "clutter". It should be used sparingly, however, since overuse may detract from the understandability of the resulting program by providing multiple names for the same item.

Examples:

```
equate Elem to Symbol_Table(I);
/* Elem has the same type as Symbol_Table(I). */
```

```
equate String to Buffer@ as character(255);
/* String is of type character(255) regardless of type of Buffer@. */
```

Like the open statement, `equate` may appear only within a procedure body, and the specified name expression is evaluated at the time the statement is performed. Subsequent changes to pointers or subscripts will have no effect.

`equate` can also be used to associate a type with an object of type `unknown` (i.e., the result of dereferencing a pointer of type `pointer to unknown`).

P. Procedure Definitions

A procedure definition contains a sequence of declarations and executable statements constituting the body of the procedure. The names of the parameters and result are determined from the type of the procedure, and are available as local variables within the definition.

The heading of a procedure definition may be immediately followed by either or both of the linkage and environment options. They may occur in either order. A semi-colon is required after the last option.

1. Linkage Option

The linkage option is used when a procedure requires a non-standard entry sequence.

System 370 Note

The entry sequence normally used by Plus/370 is compatible with the MTS coding conventions standard. This provides efficient procedure entry/exit/call but requires a stack and global storage to be set up correctly by the caller. Special linkages can be useful to establish the required set-up when entering a Plus program from a procedure that does not follow the Plus conventions.

At the time of writing, the MTS conventions are undergoing an incompatible change. The current version of the Plus compiler can be used with either the old or new forms, depending on the setting of the `%Linkage` compiler variable.

Implementation Restriction (Motorola 68000)

Plus/68000 does not implement `linkage`.

The linkage option is normally specified as part of a procedure definition but may alternatively be given with the procedure declaration.

Examples:

```
/* The following example illustrates the linkage option
   as part of the procedure declaration. */
procedure Main is
      procedure
      ...
      end linkage "PLUSENTR";
   ...
definition Main
  ...
end Main;


/* The following example illustrates the linkage option as
   part of the procedure definition. */
procedure Example is
      procedure
      ...
      end;
   ...
definition Example
   linkage "PLUSENTR";
   ...
end Example
```

The following are allowed for the linkage option:

a. `linkage "`*extname*`"`

Given a procedure heading such as

```
    definition proc
       linkage "extname";
```

where `"`*extname*`"` is a 1 to 8 character constant, the compiler generates special entry code which branches from the entry sequence of *proc* to a special linkage routine with the external symbol *extname*. The code at *extname* is expected to set up stack and global storage and then return to the entry code for *proc*. The requested stack size of the procedure, the "global id" for the procedure's environment, the size of the environment and some other data are provided in the entry sequence of *proc* and are accessible by *extname*.

If the special linkage requires non-standard exit code also, it must set up the registers in the stack in such a way that when *proc* returns, the special exit code will gain control.

The special linkage routine can be written either in Assembler or in Plus by using `linkage none` and lots of `Inline`'s.

The details of the interface between *proc* and *extname* for Plus/370 and Plus-11 are provided in Appendices D and E.

b. `linkage system`

`linkage system` requests that the compiler generate entry/exit code that is compatible with the standard "system" linkage.

System 370 Note

For Plus/370, `linkage system` allows the procedure to be called from an OS Type I ("Fortran") procedure. It is actually implemented by using `linkage "QSYSENTR"`; i.e., the special linkage capability described above. If a procedure must be called from both Plus and Fortran routines, it is necessary to use *both* the `linkage system` option (to request Fortran-compatible entry/exit code) and the type attribute `system` (see Section F–8, page 71) for the procedure type (to request Fortran-compatible calls to the procedure). You should not specify `linkage system` unless Fortran-callability is really required, since it is much less efficient than the normal Plus linkage.

PDP-11 Note

For Plus-11, `linkage system` is treated the same as the normal Plus linkage; i.e., the option is ignored.

c. `linkage none`

Given a procedure heading such as

```
definition proc
   linkage none;
```

the compiler will generate no entry code whatsoever. It is intended to make it possible—*though not necessarily easy*!—to write special linkage routines within Plus.

About the only thing you can do in a `linkage none` routine is to use `Inline` and register variables to establish the required setup. A great deal of care is required with such routines, since the compiler will assume that various registers have been set up correctly if any statements in the procedure require them. See Appendices D and E for further details.

2. Environment Option

The environment option allows a procedure to switch its global storage environment as part of the entry code.

Example:

```
procedure Example is
      procedure
      parameter Rec is pointer to Rec_Type,
      ...
      end;

definition Example
   environment Rec;
   ...
end
```

The entry code would establish the value of `Rec` as the current environment, and the type of `Rec` (`pointer to Rec_Type`) as the environment type in effect throughout the procedure. Hence `Example` could refer to fields of `Rec_Type` directly (without qualifying the references with the record pointer), and could call other procedures with environment `pointer to Rec_Type` without switching environments at the call.

The expression in the entry code may be any kind of expression returning a type allowed for procedure environments. It will usually be a parameter value (or obtained via a parameter), but might also be a procedure call, or an element of the environment provided by the caller.

Note that the caller must still provide an environment compatible with the environment type of the procedure (from the environment attribute if any, or the default `global(%Global_Id)`). This environment is in effect for the evaluation of the expression in the entry code. When setting up an environment from a parameter, as in this example, it will often be appropriate to define the procedure to have the attribute `environment unknown`, to allow it to be called from any environment.

Implementation Restriction (PDP-11)

Plus-11 does not support the mechanisms for switching global storage environments.

Q. Macro Definitions

The macro definition defines the name and parameter names of a macro. The body of the macro may be either a statement list, which becomes a separate scope-block, or a parenthesized expression. The end of a macro is indicated by either `end`, `end macro`, or either of these followed by the name of the macro.

A macro is invoked much like a procedure by use of its name followed by an argument list. The body of the macro is then substituted for the macro name, with appropriate substitutions of arguments.

During expansion of the macro, any identifiers used in it (but not declared within it) obtain the definitions in effect at the time the macro was *defined*. If an identifier was not defined at the point of the macro definition then it is a “free variable”, and will assume the definition in effect at the point of expansion. (If there is no definition in effect when it is expanded, it is simply an undefined identifier and will result in an error message.)

Example:

```
constant Svc is '0A',
   Svc_Getelt is 38;

macro Get_Elapsed_Time
   parameter is Time;
   variable Temp is Fullword in register 2;
   Inline(Svc, Svc_Getelt, Temp, 0, 1, 3); /* changes r0 - r3 */
   Time := (10 * Temp) / 3   /* convert to millisecs */
end macro
```

This macro has one parameter, `Time`. `Temp` is a local variable of the macro. The identifiers `Svc`, `Svc_Getelt` etc., refer to the definitions preceding the definition of the macro.

This macro would be used in a statement like

```
Get_Elapsed_Time(Elapsed)
```

Example:

```
macro Current_Character;
   (Substring(Str,I,J))
end Current_Character
```

This macro has no parameters. Its body is a parenthesized expression, so it is used in the context of expressions, for example:

```
Char := Current_Character()
```

Note

> Macros as described above may be removed from a future version of the language in favour of internal or "inline" procedures. We recommend that macros be used only in ways that are compatible with procedures.

R. Built-in Procedures, Constants, and Variables

Plus provides a number of built-in procedures, and a few predefined constants and variables. The names of these are predefined identifiers. The built-in definitions may be overruled by explicit declarations of the same identifiers.

1. `Address`

The `Address` procedure is used to create pointers. It takes as an argument a name expression or a constant of any type. The result is a pointer to the specified location (a value of type `pointer to ...` the argument type). When the argument is a constant, the result type is `pointer to value ...`.

2. `Alignment`

The `Alignment` procedure is used to return alignment. The first parameter may be a global block identifier, a name, or a type identifier.

Implementation Restriction (all compilers)

> Currently, `Alignment` is implemented only for global block identifiers.

When the parameter is a global block identifier, it returns the required byte-alignment factor of the global block as a number from 1 to 8. (1 means byte-aligned, 2 means halfword-aligned, and so forth).

3. `Bit_Size`

The argument of this procedure may be a type identifier, a name or a global block identifier. If a name is given, the allocated size of that name *in bits* is returned. If a global block identifier is given, the size of the global block is returned. If a type identifier is given the normal size of that type is returned. (The actual size of a variable of a given type may be bigger than the size of the type, due to padding that may be provided when variables of the type are allocated.)

The `Bit_Size` procedure is always performed at compile time.

`Bit_Size` may also be used to find the size of a variant of a record. It then has two parameters. The first is a type identifier, the second a *constant* of the type of the variant selector. The second parameter is allowed only when the first parameter is a type-identifier for a record type with variants.

If the constant specified does not match one of the variant labels, `Bit_Size` returns the size of the `else` variant, if any, or the size of the fixed part preceding the variant if `else` wasn't given.

If the second parameter is not given, it returns the size of the largest variant.

4. `Byte_Size`

The arguments and results for this procedure are the same as for `Bit_Size` (see preceding item) except that the result represents the number of bytes allocated.

5. `Code_Base_Register`

`Code_Base_Register` is implemented only by Plus/370. It is a predefined register variable of type `bit(32)`, corresponding to the register used for code addressability (normally R10). This is intended for use in special linkage routines that set up the required execution environment for Plus.

The compiler makes no attempt to interpret what you do to this register. It should be used only by experts. See Appendix D for information about Plus register usage and entry/exit code requirements.

6. `Condition`

`Condition` is implemented only by Plus/370 and Plus/68000. It is used (normally in conjunction with `Inline`) to examine the machine condition code. It accepts a single parameter which specifies which condition code settings to test for, and returns true or false according to the value of the condition code.

The parameter is currently a numeric or bit constant in the range `(0 to 15)` and is interpreted in the same way as a branch mask in an assembler branch instruction. In a future version of Plus/370, the parameter will be a `set of (0 to 3)`, specifying directly which values to test for.

Example:

```
Inline(Ltr,1,1);
if Condition(8)
then  /* condition code 0 - reg 1 was zero */
   ...
end
```

7. `Environment_Base_Register`

This is a predefined register variable of type *t*, where *t* is the environment type of the procedure referencing it. It corresponds to the register containing environment addressability (R11 for Plus/370, not implemented for Plus-11, `A4` for the Macintosh and `A6` for the AMIGA). This is intended for use in special linkage routines that set up the required execution environment for Plus.

The compiler makes no attempt to interpret what you do to this register. It should be used only by experts. See Appendices D, E and F for information about Plus register usage and entry/exit code requirements.

8. `External_Name`

This procedure may have as a parameter a procedure, global block, external variable or entry constant identifier. It returns the external (loader) name of the parameter.

9. `False`

`False` is predefined as a numeric constant with value 0.

10. `Frame_Base_Register`

`Frame_Base_Register` is implemented by Plus-11 and Plus/68000. It is a predefined register variable corresponding to the register used to address the local stack frame. In Plus-11 this variable has type `bit(16)` and is normally R5. In Plus/68000 it has type `bit(32)` and is `A6` for the Macintosh and `A5` for the AMIGA. This is intended for use in special linkage routines that set up the required execution environment for Plus.

The compiler makes no attempt to interpret what you do to this register. It should be used only by experts. See Appendix E for information about Plus register usage and entry/exit code requirements.

11. `Global_Base_Register`

`Global_Base_Register` is usually a synonym for `Environment_Base_Register` but is of type `bit(32)` (`bit(16)` for Plus-11) instead of the environment type of the procedure. On the Macintosh, it is defined to be A5, the global data base, rather than A4, the environment base.

12. `Global_Size`

`Global_Size` is a predefined constant whose value is the total size of the globals required by a program. It is a "load-time" constant (generated as a CXD), and cannot be used in situations requiring a compile-time constant.

Implementation Restriction (PDP-11)

This constant is not available for the PDP-11 compiler since *LINK11 does not support the required load-time constants.

Implementation Restriction (Motorola 68000)

This constant is not implemented in Plus/68000 since none of the linkers support the required load-time constants.

13. `High_Value`

The `High_Value` function takes as an argument a type identifier for an index type, or a name of some index type. If a name is given, the type of that name is used. It returns the highest value of the type.

14. `Inline`

The `Inline` procedure can be used to emit specific machine instructions. It is very similar to the procedure with the same name in XPL or Sue. However, Plus is aware of the format of specific machine instructions and checks that the appropriate parameters are given.

`Inline` accepts a variable number of parameters. The number and types depend on the object machine for which code is being generated, and on the specific machine instruction being emitted. In general, the parameters correspond to the required operands of the instruction, in the order that they appear *in the machine instruction* (not the order they would be specified in an assembler instruction). Some exceptions to this rule may occur, however.

Note

Using `Inline` is quite tricky. It is often advisable to turn code listing on and hand check the generated code.

a. `Inline` **for the System 370**

The first parameter of the System 370 `Inline` is always a numeric or bit-type *constant* whose value must be in the range 0 to 255. This parameter gives the op-code for the instruction to be emitted.

The subsequent parameters depend on the particular machine instruction being emitted. They may be any of the following:

•• A variable (or other name expression). Such a parameter may be used to correspond to a base/displacement pair of operands in a machine instruction. The base/displacement may also be given as two separate parameters, a register and a displacement.

•• A register local variable. This should be used for an operand that requires a register.

•• A scalar constant (any type). May be used for an "immediate" operand or for a displacement. Currently, a constant may also be used for a register operand, but this will change in a future version. In all cases, the constant must be in the appropriate range for the operand.

Note that, when coding operand lengths (for SS format instructions), the “IBM length” (actual data length−1) must be specified. `Inline` will not automatically adjust the length.

Examples:

```
constant L is '58',
   Ar is '1A',
   La is '41',
   Sla is '8B';

variable R1 is Integer in register 1,
   R2 is Integer in register 2,
   Temp is Integer;

Inline(L,R1,0,Temp);     /* L 1,Temp(0) - same as R1 := Temp */
Inline(L,R2,R1,Temp);    /* L 2,Temp(1) */
Inline(L,R2,0,R1,4);     /* L 2,4(0,1)  */
Inline(La,R1,0,0," ");   /* LA 1,C' ' = LA 1,X'40' = LA 1,64 */
Inline(Ar,R1,R2);        /* AR 1,2 - same as R1 +:= R2 */
Inline(Sla,R1,0,4)       /* SLA 1,4 -- note SLA has no index operand. */
```

(Note that the above are examples only. For most of them, it would not be necessary to use `Inline` to get the required machine instructions generated, since Plus source statements would generate the appropriate code when used with register variables.)

Warnings

The current version of the compiler attempts to ensure that the registers used in `Inline` are available, but does not currently guarantee this. In some situations, the compiler may substitute a different register for the specified one. When it does so, a warning message will be issued. When a register substitution is necessary, the compiler will make the substitution consistently though all references to it in a sequence of consecutive `Inline` instructions.

The compiler is also currently unaware of registers used by an `Inline`’d instruction but not explicitly referenced (e.g., the odd register of even-odd pairs, the intermediate registers of LM and STM, any registers required by an SVC). Such registers may be specified by the programmer by appending them as extra parameters of the `Inline` instruction, as in the next example.

Example:

```
variable Parlist is pointer to unknown in register 1,
   R14 is ... in register 14,
   R15 is ... in register 15;
Parlist := Address(X);
/* Register 1 is also required for the BALR. */
Inline(Balr,R14,R15,Parlist)
```

b. `Inline` for the PDP-11

The first parameter of the PDP-11 `Inline` is always a numeric or `bit(16)` *constant* whose value is used to determine the op-code encoding *only* for the instruction to be emitted. Any bits which are not part of the op-code should be zero.

For example, the op-code for Mov should be given as `'1000'` (although only the first digit is actually part of the op-code.)

The subsequent parameters depend on the particular machine instruction being emitted. As for the 370 version, the operands appear in the order that they occur in the machine instruction, except that the (mode, base, indexword) triplets occur together. This avoids having to include an index word if it is not specified by the mode.

For example, a move from register one to offset *disp* from register 3 would be

```
variable R1 is Integer in register 1,
   R4 is Integer in register 3;
Inline(Mov,0,R1,6,R3,disp)
```

In this example, 0 specifies the mode of the first operand (register), R1 is the register variable, 6 is the mode of the second operand (index), R4 is the register and *disp* is the offset.

The parameters for the PDP-11 `Inline` may be any of the following.

•• A variable (or other name expression). Such a parameter may be used to correspond to a (mode, base, indexword) triple.

  If the variable is a register variable, then a mode of zero is assumed. Thus the above example could also be

```
Inline(Mov,R1,6,R3,disp)
```

  The triple may also be given as two or three separate parameters, the last being omitted if index mode was not specified.

•• A register variable. This should be used for an operand that requires a register.

•• A scalar constant (any type). May be used to indicate the mode or indexword of an operand triple. A constant may also be used to specify a register for an operand, although the use of a register variable is preferred.

See also the warnings for the System 370 `Inline` regarding registers.

c. `Inline` for the Motorola 68000

The first parameter of the Motorola 68000 `Inline` is a *character string constant* giving the operation code, the size and any specific effective addressing modes. It has the format:

opcode.size mode1,mode2

The string must be entirely in lower case. See Appendix G for a list of all the recognized opcodes, sizes and modes.

The *opcode* is generally an operation code as given in the Motorola "Programmer's Reference Manual". For those "instructions" that have more than one form, such as `ADD`, the compiler defines a name for each form. Thus the compiler recognizes `"add"` as the "`ADD <ea>,Dn`" form and `"addm"` as the "`ADD Dn,<ea>`" form. Those instructions that include condition codes can be specified with any of the condition codes defined in the Motorola manual. Most 68000 instructions have one general effective address mode operand and one specific mode operand. For Plus/68000 `Inline`, you must *always* specify the general operand first. Thus the "exclusive or" operation recognized by Plus/68000 is `"eorm"` (exclusive or to memory).

The *.size* can be omitted. If the operation has no size, then it must be omitted. If the operation has a size and none is specified, then it defaults to the largest allowed by the operation.

The *mode1* and *mode2* specify either the exact addressing modes to be used, or, by their omission, that a Plus storage reference is to be used and the compiler should provide an appropriate mode. When an exact addressing mode is provided, the corresponding `Inline` operands must give exactly the parts of the mode, using index constants for parts such as displacements and register variables for registers. The parts are given in the same order as they would be specified to an assembler. For indexed modes, a constant `1` or `0` takes the place of the assembler's `.L` or `.W` (respectively).

Examples:

```
/* Do an unsigned multiply of the longwords Int_1 and
   Int_2, producing the result in Product. */
variables Int_1, Int_2, Product, Temp_1, Temp_2 are
      Integer in register;
Temp_1 := Int_1;
Inline("swap", Temp_1);
Inline("mulu.w", Int_2, Temp_1);
Temp_2 := Int_2;
Inline("swap", Temp_2);
Inline("mulu.w", Int_1, Temp_2);
Temp_1 +:= Temp_2;
Inline("swap", Temp_1);
Inline("move.w #", 0, Temp_1);
Variable Temp_3 is Integer in register;
Temp_3 := Int_1;
Inline("mulu.w", Int_2, Temp_3);
Temp_3 +:= Temp_1;
Product := Temp_3;
variable Base_Addr is pointer to unknown in register,
   Temp_Word is bit(32) in register;
/* Load the (unaligned) 4 bytes pointed to by Base_Addr
   into Temp_Word. */
inline("move.b (ar)+,-(ar)", Base_Addr, Stack_Pointer);
inline("move.w (ar)+,dr", Stack_Pointer, Temp_Word);
inline("move.b (ar)+,dr", Base_Addr, Temp_Word);
inline("swap.w dr", Temp_Word);
```

```
            inline("move.b (ar)+,-(ar)", Base_Addr, Stack_Pointer);
            inline("move.w (ar)+,dr", Stack_Pointer, Temp_Word);
            inline("move.b (ar)+,dr", Base_Addr, Temp_Word);
```

15. `Left_Justify`

This function coerces its operand to be a left-justifying (character-string-like) expression.

16. `Length`

This procedure accepts as a parameter any fixed or varying character expression. Its result is the length of the value in characters. `Length` is performed at compile-time if possible.

17. `Low_Value`

The `Low_Value` procedure takes as an argument a type identifier for an index type, or a name of some index type. If a name is given, the type of that name is used. It returns the lowest value of the type.

18. `Max`

The `Max` function takes an arbitrary number of arguments of any index type, and returns as its result the maximum of the values. The arguments must be type compatible.

19. `Min`

The `Min` procedure takes an arbitrary number of arguments of any index type, and returns as its result the minimum of the values. The arguments must be type compatible.

20. `Null`

`Null` is a predefined constant that is compatible with any pointer, procedure, or global type. It is used as a special distinguished value, for example to indicate the end of a linked list.

The value actually used to represent `Null` is 0.

21. `Offset`

If the first parameter of this procedure is a global block identifier, it returns the offset (in bytes) of that global within the global area (pseudo-register vector). This is a "load-time constant" and thus, it acts like a constant but cannot be used as a compile-time constant expression. It is equivalent to use of `Q(`*extname*`)` in an assembler program, where *extname* is the external symbol for the global.

The first parameter may also be a record type identifier. In this case, a second parameter, which must be the name of a field of the type, is also required. `Offset` then gives the offset, in bytes, of the field from the beginning of the record.

22. `Predecessor`

The `Predecessor` procedure takes as its argument a value of any index type, and returns as its result the next lower value of that type. The result is undefined if the argument is the lowest value of the index type.

23. `Program_Counter`

`Program_Counter` is implemented only by Plus-11. It is a predefined register variable of type `bit(16)`, corresponding to the PDP-11 program counter register (R7). This is intended for use in special linkage routines that set up the required execution environment for Plus.

The compiler makes no attempt to interpret what you do to this register. It should be used only by experts. See Appendix E for information about Plus register usage and entry/exit code requirements.

24. `Right_Justify`

This function coerces its operand to be a right-justifying (number-like) expression.

25. `Size`

`Size` is currently a synonym for `Bit_Size`. However, at some time in the future it will become a synonym for `Byte_Size` instead. It is *strongly recommended* that you use either `Byte_Size` or `Bit_Size` as appropriate.

26. `Stack_Base_Register`

`Stack_Base_Register` is implemented only by Plus/370. It is a predefined register variable of type `bit(32)`, corresponding to the register used to access the stack (R12). This is intended for use in special linkage routines that set up the required execution environment for Plus.

The compiler makes no attempt to interpret what you do to this register. It should be used only by experts. See Appendix D for information about Plus register usage and entry/exit code requirements.

27. `Stack_Pointer`

`Stack_Pointer` is a predefined register variable corresponding to the stack pointer register. It is implemented by Plus-11 and Plus/68000. In Plus-11, it is of type `bit(16)` and corresponds to the PDP-11 stack pointer register, (R6). In Plus/68000, it is of type `bit(32)` and corresponds to `A7`. This is intended for use in special linkage routines that set up the required execution environment for Plus.

The compiler makes no attempt to interpret what you do to this register. It should be used only by experts. See Appendices E and F for information about Plus register usage and entry/exit code requirements.

28. `Substring`

The `Substring` procedure is used to select a substring from a fixed or varying-length character expression.

The procedure takes two or three parameters. The first parameter specifies a string expression. The second parameter specifies a starting position within the string. Starting positions are zero-relative; i.e., a value of zero selects a substring beginning at the first character of the string. The third parameter, if given, specifies the length of the string to select. If it is omitted, the remainder of the string, from the specified starting position, is assumed.

The second and third parameters must be such that the selected substring lies within the string specified by the first parameter. The compiler will optionally generate extra code to run-time check the values of any substring parameters whose correctness cannot be determined at compile time.

If the first parameter of `Substring` is a name expression, and the length of the substring selected is constant then the result of `Substring` is also a name, and may be used on the left-hand-side of an assignment statement.

Examples:

```
Substring(String@,I,1) := " ";
Hex_Char := Substring("0123456789ABCDEF",I,1)
```

29. `Successor`

The `Successor` function takes as its argument a value of any index type, and returns as its result the next value of that type. The result is undefined if the argument is the highest value of the index type.

30. `True`

`True` is predefined as a numeric constant with value 1.

31. `Version`

`Version` is a predefined integer constant giving the current compiler version number in the form of $1000 * release + change$, where *release* and *change* are as described in Chapter IV, page 121.

S. Compile-Time Statements

Plus provides compile-time if statements, compiler variables and compiler procedures.

A compiler variable or compiler procedure is a predefined identifier beginning with `%`. (Normal identifiers may not contain `%`, so compiler variables and procedures cannot be confused with normal ones.) Compile-time if statements are also flagged with `%`.

1. Compile-Time If Statements

The compile-time if statement allows conditional compilation of program segments. It is syntactically just like a regular if statement, except each keyword is preceded by `%`. The expression in the `%if` part must evaluate to a constant at compile time, which is used to determine whether the statements in the `%then` part or the `%else` part are included in the program.

The end of the `%if` statement may be indicated by `%end`, `%end %if` or `%end if`.

Example:

```
constant Debugging is True; /* Set False for production use. */
    ...
%if Debugging
%then
    /* Do this only when debugging. */
    ...
%end %if
```

In this example, the statements between `%then` and `%end %if` are included in the program, since the constant expression `Debugging` is true. If the constant declaration is changed to

```
constant Debugging is False
```

then the statements will be skipped.

The `%if` statement may appear anywhere a statement is allowed in Plus, inside or outside of procedures or global blocks. The `%then` part or `%else` part is included in the enclosing list of statements and must be appropriate for its context. That is, if the `%if` statement is not inside a procedure, then the statement list it contains must consist only of declaration statements (or other compile-time statements).

The statement list in the `%then` part or `%else` part must be syntactically valid even if it is skipped. However, skipped statements may include references to undeclared variables and other "semantic errors" without complaint from the compiler.

Note that procedure definitions and global blocks are syntactically allowed as statements, so that entire procedures or global blocks may be included or skipped by a `%if` statement.

A sequence of nested `%if`'s can be combined using `%elseif` as with regular if statements.

Example:

```
%if %Installation = "UBC"
%then
   constant Site_Name is "University of B.C."
%elseif %Installation = "UM"
%then
   constant Site_Name is "University of Michigan"
%elseif %Installation = "SFU"
%then
   constant Site_Name is "Simon Fraser University"
%else
   constant Site_Name is "?"
%end %if
```

In this example, the declaration of `Site_Name` is selected according to the value of `%Installation`.

Note that a normal if statement is quite different:

```
if %Installation = "UBC"
then
   constant Site_Name is "University of B.C."
...
end if
```

is an executable statement and is only allowed inside a procedure, while the `%if` may go anywhere. In any case, the body of the if statement is a separate scope block, so the declarations it contains are discarded at the end of the if statement, while the declarations in a `%if` statement become part of the scope containing it.

2. Compiler Variables

Compiler variables are used to set various compiler options, and to access their values. A compiler variable may appear on the left-hand-side of an assignment statement. The right-hand-side must be a constant (or constant expression) of an appropriate type. This value becomes the new value of the compiler variable while compiling the remainder of the program (or until changed again).

A compiler variable may also appear in any context where a constant is allowed. It is always replaced by its value at that point in the compilation.

Example:

```
%Title := "This is the way the title is set";
%List := True;

Message(M, "Error at coordinate <i> in procedure || %Current_Procedure
   || "</>", %Coordinate)
```

Some of the options affect code generation (e.g., run-time checking). In general, the code generated will be determined by the values of the options *at the end* of the procedure. You cannot have an option on for parts of the code in a procedure and off for other parts.

The list of available compiler variables is implementation-dependent. The following are those which are defined by the current compilers. Except as noted, they are implemented for all compilers.

`%Assertion_Check := {True|False}` *default*: `True`

If `%Assertion_Check` is true, then code is generated to check the expressions in any assert statements. If it is false, assert statements are treated as comments.

`%Assign_Check := {True|False}` *default*: `False`

`%Assign_Check` is intended to check for assignments in which the destination is used as part of the source, and is changed before it is referenced. For example:

```
X := Y || X
```

This error is not currently detected except in one or two special cases.

`%Check := {True|False}` *default*: `True`

When a new value is assigned to `%Check`, each of `%Range_Check`, `%String_Check`, `%Assign_Check`, `%Assertion_Check` and `%Stack_Check` is automatically reset to the same value. Thus `%Check := False` may be used to turn off all run-time checking.

`%Compile := {True|False}` *default*: `True`

If this option is set off, the compiler will perform syntax checking only. It will not perform any other error checking or compile-time processing and will not generate object code. Once set to `False`, the option cannot be reset during the run (since subsequent assignments to compiler variables are not processed).

The compiler will still produce a paragraphed copy of the source if requested.

`%Compiler_Dumps :=` *n* *default*: `1`

This option controls the printing of linkage trace-backs and storage dumps if a program interrupt occurs in the compiler. It is primarily of interest to the compiler implementors.

`%Compiler_Debug :=` *n* *default*: `0`

This option controls various internal compiler debugging options. It is of interest only to the compiler implementors.

`%Convert := {True|False}` *default*: `False`

This option requests the compiler to convert the paragraphed copy to adjust for incompatible changes to Plus that may have occurred.

The exact effect may vary from time to time. Currently, the actions performed are:

1) Any symbol that has been `%Unreserve`'d will be converted to a valid identifier by appending a "`#`" in the copy produced on unit 1. Thus, for example, setting `%Convert := True`, in conjunction with the compiler procedure `%Unreserve("entry")`, will produce a paragraphed copy in which all occurrences of `entry` are replaced by `Entry#`

2) Any uses of the built-in procedure `Size` will be converted to use `Bit_Size` instead.

`%Coordinate`

Contains the source-coordinate of the current line, as an integer. This may be useful in producing error messages for debugging purposes.

`%Current_Procedure`

Contains the name of the procedure currently being compiled. This may be useful in producing error messages for debugging purposes.

`%Date`

Contains the date at the start of compilation as a character string in the form "day mon dd/yy".

`%Dump_Tree := {True|False}` *default*: `False`

If true at the end of a procedure, the intermediate code tree is printed. This is primarily of interest to the compiler implementors.

`%Entry := "`*string-constant*`"` *default*: `""`

This option is ignored by Plus-11.

For Plus/370, the specified character string is used as the name of the entry point of the program and is punched in an ENT record at the end of the compilation. If it is a null string, `""`, as it is by default, no ENT record is produced. (However, the standard Plus/370 library definition of `Main` sets `%Entry` to `"MAIN"`.)

For Plus/68000 with `%Target_Operating_System` of `"MAC/MPW"`, if a procedure whose external name matches the string specified is defined, then it is marked as being the entry point of the program.

`%Footer := {True|False}` *default*: `True`

If this is set false, footer lines (using carriage control '`<`') will not be printed in the source listing.

`%Installation` *default*: *installation dependent*

This is intended to assist people writing programs that are used at more than one MTS installation, but that must contain installation-specific code. The value is initialized to the CNFGINFO "share code" field (for example, `"UBC"` at UBC, `"UM"` at the University of Michigan, etc.). Note it is a compile-time value—it reflects the CNFGINFO code at the time a program is compiled, *not* at the time it is executed.

This may be tested in if statements or compile-time `%`if statements to select between installation-dependent alternatives.

`%Installation` may be assigned another value to test out the compilation of alternate versions. For example:

```
Run *Plus ... Par=%Installation:="SFU"
```

would compile the "SFU" version of a program.

`%Global_Id := bit(32)-`*constant* *default*: `"PLUS"`

`%Global_Id` is implemented only by Plus/370and Plus/68000. It specifies the "global-id" for the global storage type containing all global variables in the program. See Section C–3, page 46.

`%Instruction_Set := "`*string-constant*`"` *default*: Plus/370—`"STANDARD"`
Plus-11—`"EXTENDED"`
Plus/68000—`"STANDARD"`

This option may be used to specify the instruction set available on the object machine. The possible values that may be specified currently are `"STANDARD"`, `"BASIC"`, or `"EXTENDED"`. Currently it is completely ignored by Plus/370 and Plus/68000.

For Plus-11, the option `"STANDARD"` may be used to generate code for a machine that doesn't have the `Mul`, `Div`, and `Ash` instructions, and the option `"BASIC"` may be used if the machine also doesn't have the `Sob` instruction.

`%Library := {True|False}` *default*: `False`

This option is ignored by Plus-11 and Plus/68000. For Plus/370, it controls whether special loader records are output at the end of the object module. These records are needed to access the resident system Plus library routines. `%Library` defaults to false (the records are not punched). (However, the standard Plus library definition of `Main` also sets this option to `True`.) See Chapter IV, page 127 for information about the loader records required to run a Plus/370 program.

`%Lines_Per_Page :=` *n* *default*: `60`

This option sets the number of lines that the Plus paragrapher will put onto a page of the listing.

`%Linkage := "`*string-constant*`"` *default*: Plus/370—`"NEW"`
Plus-11—`"ALTERNATE"`
Plus/68000—`"NEW"`

This option controls some details of the procedure linkage assumed by the compilers.

The allowed values are currently `"OLD"` or `"STANDARD"` (which are synonyms) and `"NEW"` or `"ALTERNATE"` (which are also synonyms).

For Plus/370, `"OLD"` or `"STANDARD"` means the old (pre-1986) form of the MTS coding conventions is to be used. `"NEW"` means the new ("1986") conventions are to be used. The default[8] is `"NEW"`.

For Plus-11, the use of `"ALTERNATE"` reverses the compiler's use of `R4` and `R5`. See Appendix E.

Plus/68000 ignores this option.

`%List := {0|1|2}` *default*: `1`

This option controls the source listing. If set to 0, no source listing is produced. If set to 1, source from `Scards` is listed, but any input included from a library is not listed. If set to 2, all input is listed.

The paragraphed source copy on unit 1 is produced (if unit 1 is assigned) independently of the setting of `%List`. The copy will never include input from a library.

`%List_Code := {True|False}` *default*: `False`

If this is true at the end of a procedure a listing of the object code for that procedure is produced.

`%List_Code` may also be assigned the value 2 or 3, which cause the intermediate code representation and associated tables to be dumped. This information is probably of interest to the compiler implementors only.

[8] As of July 1987. This is likely to change to `"STANDARD"` when the meaning of `"STANDARD"` is changed to be synonymous with `"NEW"`.

`%Listing_Character_Set := "`*string-constant*`"` *default*: `"MIXED"`

This compiler variable can be used to indicate to the compiler what characters are available on the listing device. This is only a hint—the compiler will not necessarily adhere. The values currently allowed are:

`"MIXED"`	indicates upper and lower case may be used.
`"UPPERCASE"`	indicates only upper case letters are available. The compiler will *not* translate everything to uppercase when this option is selected; it just doesn't bother converting various things to lower case.
`"TN"`	means the IBM TN character set may be used. Currently, this just causes use of TN box corners and edges for `/*BOX ...` comments.

`%Lower_Case := {True|False}` *default*: `False`

If this is set true, the source listing and paragraphed copy will be produced with all keywords and identifiers converted to a standard upper-and-lower case format. String constants and comments will be left in their original case.

`%Lower_Case` may also be assigned the value 2, in which case comments will be converted to all lower-case.

`%Merge_Unref := {True|False}` *default*: `True`

controls whether the cross reference listing of unreferenced identifiers appears as a separate listing. If it is true at the end of compilation, then the listing of any unreferenced symbols is merged in with the regular listing. If it is false at the end of compilation, unreferenced symbols appear as a separate cross reference listing. Note that in either case, whether an identifier with no references appears or not is controlled by the setting of `%Unref` when the identifier was declared.

`%Object_Length :=` *number*

The value of `%Object_Length` at the end of a procedure determines the maximum length of the object module records that will be punched for that procedure.

For Plus/370, by default, *number* is the same as the maximum output record length of the file or device assigned to SPUNCH. It may not be set to a value less than 40 or greater than the maximum length of the output device.

This option should not be set bigger than 255 with Plus-11, since `*Link11` doesn't support long object records.

This option is ignored by Plus/68000.

`%Optimize := {0|1|2|3}` *default*: `0`

This option is currently unimplemented. It will be used to select the kind of optimization wanted.

`0` indicates no optimization.

`1` indicates optimization for a "reasonable combination" of space and speed.

`2` means optimize for space.

`3` means optimize for speed.

`%Page_Width :=` *n* *default*: `132`

This option sets the page width that the paragrapher uses to produce the listing.

`%Preempt := {True|False}` *default*: `True`

By default, when the compiler runs out of general registers during code generation, it will continue by storing out some of the registers in use and restoring them when necessary. If `%Preempt` is set to false, the compiler will not generate this register preemption code, but will abandon code generation if it can't compile a procedure without any register preemptions.

This option is mainly used with `linkage none` routines, for which preemption code may not be safe (because the stack may not be set up at all times). In such procedures, it may be preferable to detect that the procedure required preemptions via the resulting error message.

As with other compiler options affecting code generation, the value in effect at the end of the procedure applies to the whole procedure. You can't set this option off for only part of a procedure.

This option isn't supported by Plus-11 or Plus/68000.

`%Productions := {True|False}` *default*: `False`

If this compiler variable is assigned the value true, then a line will be printed giving the number of each syntax production as it is applied during the parsing of the program. This output is primarily of use to the compiler implementors.

`%Range_Check := {True|False}` *default*: `True`

If `%Range_Check` is true, the compiler will generate extra code for assignments and array subscripts to check that the value is within the declared range of the variable or array index. A run-time check will not be generated if the compiler is able to determine at compile time that the value should be within the declared range. For example, when assigning a variable to another with the same range, no run-time check is performed.

The range-checking facilities will sometimes catch uninitialized variables, but cannot be relied on to do so.

Checking may be disabled by assigning `%Range_Check` the value false.

`%Regression_Test := {True|False}` *default*: `False`

This option alters the output of the compiler to make it more independent of the compiler version number and the time of compilation. It is intended to allow comparison of the output of different versions of the compiler. This is primarily of interest to the compiler implementors.

`%Segment := "`*string-constant*`"` *default*: `"Main"`

This option is only implemented by Plus/68000 and is ignored unless `%Target_Operating_System` is set to `"MAC/MPW"`. This option sets the loader "segment name" for the following procedures.

`%Source_File`

Contains the current source file name as a character string. This may be useful in producing error messages for debugging purposes.

`%Source_Line`

Contains the MTS line-number of the current source line, in internal form as an integer. This may be useful in producing error messages for debugging purposes.

`%Stack_Check := {True|False}` *default*: `True`

This option is only implemented by Plus/370. If `%Stack_Check` is true, and `%Linkage` is `"NEW"`, the compiler will generate code as part of the entry sequence to check for stack overflow. The option is ignored if `%Linkage` is `"OLD"`.

`%Statistics := {True|False}` *default*: `False`

If `%Statistics` is true at the end of the input to the compiler, a number of messages will be printed describing the use of various compiler tables, and the values of various counters.

This information is primarily of interest to the compiler implementors.

`%String_Check := {True|False}` *default*: `True`

If `%String_Check` is true, the compiler will generate code to check for string assignments in which the source is longer than the destination, and to check for `Substring` functions in which the designated substring does not lie within the string.

`%Subtitle := "`*string-constant*`"` *default*: *none*

Sets a subtitle to be printed on the third line of each page.

`%Target_Machine` *default*: Plus/370—`"IBM/370"`
Plus-11—`"PDP-11"`
Plus/68000—`"MC68000"`

This option contains a string describing which compiler is being used. It may be useful in conditional compilation statements to isolate machine dependent statements.

`%Target_Operating_System :=` *string-constant* *default*: Plus/370—`"MTS"`
Plus-11—`"UBCNET"`
Plus/68000—`"MAC/MPW"`

This option specifies the system that the code is to run on. It may affect code generated, particularly for procedure calls and the implementation of loader objects. See the Index for more details.

For Plus/370, the possible values are `"MTS"` and `"MVS"`.

This option is ignored by Plus-11, and may be set to any string.

For Plus/68000, the possible values are `"MAC/MPW"`, `"MAC/MDS"` and `"AMIGA"`.

`%Test := {0|1|2}` *default*: `1`

For Plus/370, if this variable is non-zero, the compiler will generate SYM records as part of the object deck produced, to assist in debugging the object program. If the value is 1 (`True`), the object program can be used with either the “current” or “new” versions of SDS. If it is 2, it can be used with the “new” version of SDS only. See Chapter IV, page 127 for details of the debugging information produced.

For Plus-11, it causes some information about variable and record offsets to be “dumped” in the listing. This information may assist with debugging.

Plus/68000 currently ignores this option.

`%Time`

Contains the time-of-day at the start of compilation in the form `"hh:mm:ss"`.

`%Title := "`*string-constant*`"` *default*: *compiler version etc.*

Sets the title to be printed on the first line of each page.

`%Unref := {True|False}` *default*: `True`

Controls printing of the cross reference for identifiers that are never referenced. If true when an id is declared, then it appears in the cross reference even if there are no references. If false when an id is declared, that symbol will appear only if there is at least one reference.

Note that the effect is determined at the point of the declaration of a symbol. This means it is possible, for example, to set `%Unref := False` before including library declarations, so that included symbols which are not used don’t clutter up the cross reference.

`%Xref := {0|1|2}` *default*: `2`

Controls how much information is entered in the cross reference. If it is set to 0, then nothing in entered in the cross reference. If it is set to 1, then declarations will be entered, but references to the declared identifiers will not be reported. If it is set to 2, then declarations and all references will be reported.

Note that the information collected for a given identifier is determined at the point of the declaration of the identifier. That is, if the setting is 1 when a variable is declared, then the declaration will be entered in the cross reference, but references to the identifier will not be collected, even if `%Xref` is subsequently changed to 2.

`%Xref_Scope := {0|1|2|3}` *default*: `3`

Controls what identifiers are included in the cross-reference. If it is set to 0, no identifiers are entered. This is the same effect as `%Xref := 0`. If it is set to 1, only external symbols (procedures, globals, external variables) are entered. If set to 2, only global symbols (everything defined in global blocks or external to procedure definitions) are entered. If set to 3, all identifiers are included.

3. Compiler Procedures

Compiler procedures may appear syntactically anywhere a normal statement might appear. The effect depends on the specific procedure invoked.

Example:

```
%Eject();
%Include(Integer, String_type)
```

The list of available compiler procedures is implementation-dependent. The following are those which are defined by the current compilers. Except as noted, they are implemented for all compilers.

`%Double([`*n*`])`

Causes the next output line to be preceded by a skip to a "double" page. `%Double(1)` ejects to a page with an odd page number (a "front" page). `%Double(2)` ejects to an even ("back") page. With no parameter, it currently behaves like `%Double(2)` at UBC. This may change, however (and may differ at other installations).

`%Dump()`

This procedure dumps the contents of various pass 1 compiler tables as they exist at the point where `%Dump` occurs. This information is primarily of interest to the compiler implementors.

`%Eject()`

Causes the next output line to be preceded by a page skip.

`%Include(`*identifier*`,...)`

`%Include` is used to conditionally include members of the source libraries, as described in Chapter VII, page 135.

`%Map(`*name*`,...)`

`%Map` can be used to obtain a storage layout map for a record type. It requires one or more parameters, which must be type-identifiers or names of a record type.

This procedure is intended to help in ensuring that a Plus declaration correctly reflects a corresponding assembler dsect. It produces a listing of all fields of the record, giving the "access-address" and a hexadecimal mask indicating which bits are accessed by the field name. It will follow nested record types to a depth of 5 levels.

`%Message("`*string-constant*`",...)`

`%Message` outputs the given string constants to the source listing, and to `Sercom` if it is different from `Sprint`. Each line is flagged with "`*** Message`".

`%Mts()`

Causes the compiler to return immediately to the operating system, in a `RESTART`-able way.

`%Pop(`*compiler-variable*`,...)`

`%Pop` is used to restore the value of a compiler variable previously saved with `%Push`. If there is no stacked value for the specified compiler variable, the initial default value is restored.

`%Print("`*string-constant*`", ...)`

`%Print` outputs the given string constants to the listing file, one line per string. This is useful for outputting listing control lines. These lines are *not* examined, altered or counted by the paragrapher.

`%Punch("`*string-constant*`", ...)`

`%Punch` outputs the given string constants to the object file produced by the compiler (one string per record). This is useful for outputting `$Continue with` lines, or auxiliary loader control records.

`%Push(`*compiler-variable*`,...)`

`%Push` may be used to stack the values of any compiler variables. The compiler procedure `%Pop` is used to restore the value. For example:

```
%Push(%Title); /* save current title */
%Title := ...
   ...
%Pop(%Title) /* restore saved title */
```

A list of compiler variables may be pushed in a single use of `%Push`.

`%Unreserve("`*string*`",...)`

This compiler procedure can be used to indicate that the specified strings are not to be treated as reserved words for the remainder of the compilation. This is intended to allow programs written before the addition of new reserved words to continue to compile without other changes. For example,

```
%Unreserve("value","reference","entry")
```

would cause these reserved words to still be treated as a identifiers. If this is used, however, the facilities implemented by the keywords will not be available.

Note that `%Unreserve` can be used in conjunction with `%Convert` to produce a copy of the program in which the reserved words have been converted to harmless identifiers.

IV. Using the System 370 Plus Compiler

A. Compiler Versions

The current stable version of the Plus/370 compiler is found in the file `*Plus`.

The file `Plus:Plus>` contains the latest version for testing. The file `Plus:Plus<` (when it exists) will contain a backup version of the compiler. This version will normally exist only after major changes.

At UBC, the file `Plus:Plus#` will always contain the most recently distributed version of the compiler. Any programs to be distributed to other installations should be compiled with this version to ensure they do not depend on new features or bug fixes.

Each version of Plus has a version number (which appears in the default title, and in the object module END record). The version number is of the form '*n*–*m*.' '*n*' is the release number; it is incremented by one each time the compiler is completely regenerated. '*m*' is the change number, incremented for each change installed.

All compiler changes are described in the `*Forum` conference "Plus-Internals". New features and incompatible changes are also announced in the `*Forum` conference "Plus".

B. Compiling a Program

The compiler is invoked with an MTS `Run` command of the following form:

`Run *Plus [`*logical-units*`] [Par=`*statements*`]`

The following logical units may be specified on the `Run` command:

`Scards` Specifies the file or device containing the source program. Input records must not be longer than 255 characters.

`Sprint` The paragraphed listing is produced on `Sprint`. See below.

`Sercom` Error messages and certain other messages written to `Sprint` are echoed to `Sercom` if `Sprint` and `Sercom` do not refer to the same file or device.

`Spunch` The object module is produced on `Spunch`. If `Spunch` is not specified it defaults to the file `-Load`.

If `Spunch` specifies a temporary file, it will be emptied automatically before use. If `Spunch` specifies a permanent file, the file must be emptied by the user before running the compiler.

If a file (either permanent or temporary) is specified with a line-number range, then the specified range must be empty, but the whole file does not need to be empty. The compiler will not use the file if it already contains lines in the specified range.

`0` Unit 0 is used to specify a source library or libraries. If it is not specified, the default library (`*Plus.Sourcelib`) is assumed. This library contains a number of useful standard definitions, including declarations of many of the MTS system subroutines. Documentation of the members of `*Plus.Sourcelib` appears in the writeup **UBC PLUS LIBRARY**.

Note if unit 0 is specified, it is used instead of `*Plus.Sourcelib`. If it is intended to use both, `*Plus.Sourcelib` must be concatenated to the private library.

See Chapter VII for details of library format.

`1`	If unit 1 is specified, it will be used for a paragraphed copy of the source suitable for use as input to the compiler. If unit 1 specifies a temporary file, it will be automatically emptied. If it specifies a permanent file, the file must be emptied before running the compiler.
`2`	If unit 2 is specified, it will be used for a machine-readable log of the errors in the source.

The `Par=` field may specify any valid Plus statements. This is passed to the compiler (followed by a terminating semicolon), as the first input record to be processed. The `Par=` field is normally used in this way as a means of specifying the initial settings of compiler options.

The compiler passes back a return-code in R15. This is set as follows:

0—no errors or warnings detected.
4—warnings but no errors detected.
8—errors detected.

The return code may be tested by MTS command macros.

C. Compiler Output

1. Source Listing

The source listing is produced paragraphed according to precise paragraphing rules, intended to clearly indicate the control structure of the program. Source listing may be turned on and off with the `%List` compiler variable, described on page 114.

To the left of the source listing are two columns of numbers. The first column contains the input line number corresponding to the text on the line. The second contains a "source coordinate" which is used in compile-time and run-time error messages to indicate the point of the error. The source coordinate is reset to 1 for each procedure and global block and each macro definition. It is incremented for each "paragraphed line". It is not incremented when a paragraphed line is split across two printer lines as a result of the paper width limitation. The source coordinate is also used in the SYM records generated by Plus for use with SDS (see Section F, page 127).

The input file name appears to the right of the listing each time it changes, and on the first source line of each page.

A blank line appears in the listing wherever (and only where) one appears in the input.

Comments are normally formatted in the output with one blank between each "word". The options `frame`, `box`, `as_is`, and `centre` may be specified to control the formatting of the comment—these are described below. If a comment is the first thing on an input line, or if it is to be framed, it will begin a new line in the output.

Certain annotations appear in the source listing. Each exit, repeat or return is marked with "`...`" to indicate the level of the compound being exited. A heading appears at the beginning of a procedure definition, specifying the names of the parameters and result of the procedure. (This is because the procedure declaration in which they are specified may be elsewhere in the listing.)

Titles and subtitles in the listing may be set by the `%Title` and `%Subtitle` compiler variables. By default, the title specifies the compiler version and user id.

The compiler produces footer lines indicating the procedures and global blocks defined on each page. This footer may be turned off (e.g., if output is intended for a printer that does not support footers), by means of the compiler variable `%Footer`.

2. Comment Paragraphing

The comment start symbol `/*` may be immediately followed (with no intervening blanks) by one or more of the options `frame`, `box`, `as_is` or `centre`(or `center`) (separated by commas if more than one appears).

If the option `frame` appears, the listing of the comment will be surrounded by a frame of "`*`"s.

If the option `box` appears, the listing of the comment will have a line-box drawn around it. If the `%Listing_Character_Set` compiler variable is set to `"TN"`, the box will use the TN box characters; otherwise it will use characters from the PN character set.

If the option `as_is` appears, the comment will be output "as-is", with horizontal spacing preserved from the compiler input. The entire comment will be moved left or right to line up the "`/*`" with the current indentation level, but internal blanks will be preserved, and successive lines of the comment will be moved left or right as necessary to maintain the same relative position.

If the option `centre` (or `center`) appears, then the lines of the comment will be centred in the output. Each input line generates one line of centred output. `as_is` is ignored if `centre` is specified.

The words `frame`, `box`, `as_is`, or `centre` themselves do not appear in the listing.

A new line is started in the listing following any comment. A null comment ("`/**/`") is suppressed in the listing, but still causes a new line. Hence it may be useful in some situations where the line-breaks determined by the paragrapher are not adequate.

For example, if the input is

```
if Substring(Symbol, 0, Symbol_Length) = Test1 /**/
   or Substring(Symbol, 0, Symbol_Length) = Test2
then
   ...
end if
```

the paragraphed listing will appear as

```
if Substring(Symbol, 0, Symbol_Length) = Test1
      or Substring(Symbol, 0, Symbol_Length) = Test2
then
   ...
end if
```

Without the `/**/`, the parapgrapher would fit some part of the second line onto the first, and break the expression at a less appropriate place.

In a similar way, the sequence "`*//*`" within a comment is suppressed from the listing, but still causes a new line. New comment options may follow the "`/*`". However, if a frame or box is in effect, it will continue to the final end-of-comment.

For example,

```
/*box,centre
         Linear and Binary Searching
*//*
   This example program demonstrates...
*/
```

The entire comment will be surrounded by a "box" frame, but only the first part of the comment will be centred.

3. Paragraphed Copy

If unit 1 is specified on the `Run` command, a paragraphed copy suitable for use as input to the compiler is produced. The paragraphed copy is in most respects the same as the listing; however, it is intended to be a more exact duplicate of the input than the source listing.

The annotations added to the listing do not appear in the paragraphed copy. Input which is included from a library (via `%Include`) does not appear in the paragraphed copy. (The `%Include` statement is echoed however.)

The comment options and the sequence `/**/` or `*//*` are copied across to the output. A frame or box will not appear around the paragraphed copy of a comment.

The maximum length of an output line in the paragraphed copy is 68 (for convenient full-screen editing), while in the source listing it is 90. There is currently no way to change these lengths.

4. Cross-Reference

The source listing is followed by a cross-reference of identifiers used by the program.

The exact contents of the cross-reference are controlled by a number of compiler variables. The option `%Xref` controls how much is reported for each identifier. The options `%Xref_Scope` controls which identifiers are included in the cross reference. The options `%Unref` and `%Merge_Unref` control printing of identifiers that are defined but never referenced. See the descriptions of these compiler variables for details.

Entries in the cross reference indicate the general class of identifier (constant, variable, type, etc.). References are given in the form '$p : c_1, c_2, ..., c_n$', where p is a page number in the listing and $c_1, c_2, ..., c_n$ are source coordinates of references on that page. (Since coordinates start over for each procedure, and there may be more than one procedure on a page, this is not necessarily a completely precise reference.) Each reference coordinate may be followed by a one-character code indicating whether the program stores, dereferences, the symbol, etc., at that line in the program. A key for the codes used appears at the beginning of the cross-reference.

5. Errors

If an error occurs during pass 1 of the compilation, the current line is output immediately, followed by the error message. The line listed will always contain the current input line number and file name. Errors encountered after pass 1 cause a message to be issued at the end of the procedure listing.

In all cases, error messages (and the current line if any) are also echoed to `Sercom`, if `Sprint` and `Sercom` are different. Whenever an error message is issued, the flag `*** errors ***` is placed in the bottom right-hand corner of the listing for the next two pages. This helps find the error messages in a large listing.

If unit `2` is specified on the `Run` command, any errors will also be recorded in the specified file. This file can be used to automatically step through the errors with an editor.

D. Running A Plus/370 Object Program

If the main procedure for a Plus/370 program is defined by including the library definition for `Main`, and the object file is defaulted to `-load` when the program is compiled, the program can be executed with a `Run` command of the form:

`Run -load ... [Par=`*options*`;` *user-par*`]`

The library definition of `Main` specifies the special linkage routine `PLUSENTR` is to be used and causes required loader control records to be emitted at the end of the object file.

The special linkage routine `PLUSENTR` may process certain options from the `Par` field of the `Run` command before it passes control to the main procedure.

If the procedure declaration for the main procedure specifies a `stacksize` option, or if a loader record has been added to define the external symbol `STAKSIZE`, then `PLUSENTR` ignores *all* the options in the `Par` field.

The following options will be processed by the default linkage routine `PLUSENTR` when the stacksize has not been specified.

`STACK={`*n*`|`*n*`B|`*n*`K|`*n*`P}`	specifies the amount of memory to allocate for a stack. By default, one page is allocated. Specifying `STACK=`*n*`K` allocates *n* 1024 byte blocks, *n*`P` or just *n* specifies the size in 4096 byte pages while *n*`B` is the size in bytes.
`HIGH_WATER`	will cause `PLUSENTR` to output a message at the end of execution which gives the amount of stack allocated and the amount changed during execution.

The procedure `Main`[1] may optionally be declared to have a parameter and return-value, as follows:

```
procedure Main is
      procedure
      reference optional parameter Par is character(0 to 256) in
            register 0
      result Rc is Integer in register 15
      end linkage "PLUSENTR"
```

[1] It is not required that your main program be called "`Main`". You can use any identifier, provided you specify `linkage "PLUSENTR"`.

This is the declaration that will be included from the standard source library if you specify `%Include(Main)`.

To define the stacksize to allocate as part of the declaration, this should be changed to

```
procedure Main is
      procedure
      ...
      end linkage "PLUSENTR" stacksize n
```

where *n* is an integer constant for the size (in bytes) you want.

When `Main` is called, it will be passed as a parameter that part of the `Par=` field following the semicolon. The value returned by `Main` will be set in `R15` as a return-code from the program.

The stack size is increased to the nearest page (4096 byte) multiple. When `%Linkage="OLD"` is in effect, an extra page is also allocated at the end. This extra page is protected so that it will cause a protection exception if a program attempts to use storage beyond the end of the stack. Thus a protection exception in a Plus program (especially if at the entry sequence or a procedure), may be an indication that a larger stack is required. The only time you might not get a protection exception is if you have a large but unused variable on the stack, and so "hop over" the stack fence. If `%Linkage="NEW"`, the stack fence is not allocated, since the compiler option `%Stack_Check` can then be used to implement stack overflow checking.

The run-time checks performed by Plus are implemented by causing a program interrupt which is interpreted specially by the program interrupt handler set up by `PLUSENTR`. This normally dumps the registers and a limited amount of memory and provides a trace back following any program interrupt. If the program is being executed under control of SDS, however, this information will not be given. Instead just a message (for Plus run-time checks) is displayed, then the interrupt handler returns to SDS for further processing.

It is not necessary to use the run-time support provided by `linkage "PLUSENTR"`, provided the required stack and global storage environment are set up before an Plus procedure is executed. See Appendix D for further details of the requirements. Note that if the normal program interrupt handler is not used, the program must either be prepared to handle for itself any program interrupts resulting from run-time checks, or must be compiled with all run-time checks disabled.

E. Loader Records Required By Plus Programs

In order to load correctly, a Plus object file must contain some special loader records to interface with procedures and global variables defined in the resident system. These records should normally be at the end of the object file. The usual loader records are generated automatically at the end of the object deck if the option `%Library` is true. (This option defaults to false, but is set true if `Main` is included from `*Plus.Sourcelib`.) However, for programs compiled in pieces and later combined, it may be necessary to add the records "by hand".

1. `%Linkage="OLD"`

If `%Linkage` is `"OLD"`, the following record is normally required:

```
$Continue With Old:OldCCLib
```

This record causes the inclusion of the old version of the runtime library. Alternatively this library could be specified on the MTS `Run` command:

```
Run -load+Old:OldCCLib ... [Par=options; user-par]
```

2. `%Linkage="NEW"`

If `%Linkage` is `"NEW"`, the following records are normally required:

```
column →  2    7         17
          RIP  QLCSPR
          RIP  CCSYMBOL
          LCS            LCSYMBOL
          LCS            CCSYMBOL
          LCSPR          QLCSPR
```

The LCSPR record for `QLCSPR` is required so that the global (PRV) variables used by the program being run will extend the global storage used by the normal Plus entry routine `PLUSENTR`. The new coding conventions require that the first two words of the global storage be specially defined. This LCSPR for `QLCSPR` ensures the required setup. An LCSPR for `QGLOBAL` may be substituted to define only the first two words, if the program does not use `PLUSENTR`. The program will not execute correctly if neither `QGLOBAL` nor `QLCSPR` is used.

The LCS record for `CCSYMBOL` is required to tell the loader to search the symbol table `CCSYMBOL`, in order to find any Plus library routines used. If the program doesn't use any of the Plus library routines, this record may be omitted.

The other records are needed to tell the loader to find the definitions of `QLCSPR` and `CCSYMBOL`.

A copy of these records is in the file `Plus:Endjunk`, which can be copied to the end of the object program.

F. Debugging Plus Programs

If the compiler variable `%Test` has the value 1 (as it does by default), the compiler will generate SYM records to assist in debugging the Plus program under SDS. If it has the value 2, the SYM records will be generated for use with a new version of SDS.[2]

The standard Plus run-time support sets up a program interrupt exit to intercept program interrupts within the program. When a program is run under control of SDS, this default interrupt handler will return to SDS with the state at the time of the interrupt intact; thus SDS commands can be used to explore the problem. Note that run-time errors (range checks, assertion failures, etc.) in Plus programs are signalled by a program interrupt (an operation exception with an operation code of zero). For these interrupts, the interrupt handler will decode the check condition and output an appropriate message before returning to SDS.

If the program is to be restarted following a run-time error intercept, the command

```
GO $PSW+2
```

should be used rather than `CONTINUE`.

[2] You may need to issue the MTS command `Set Version(Sds)=New` to use the new version.

The support provided for debugging Plus with SDS is still rather rudimentary, since SDS is not prepared to cope with many of the basic concepts of Plus (such as programmer defined types).

Currently, SDS symbolic information is generated as follows:

1. Source coordinates are emitted for all lines of the source for which there is generated code. These are referenced with symbols of the form `#`*n* (for source coordinate *n*).

2. The procedure name (Plus identifier) is emitted as a label at the beginning of the code csect.

3. A dsect is generated for each global block. The dsect name is the external name of the global (first four and last four characters of the name).

SYM information is generated for all variables in the global. In order to reference them, SDS must be told where the global is based. This requires two steps:

a. Specify the base of the pseudo-register with the SDS command

```
USING PRAREA $GR11
```

at any time after the pseudo register vector is allocated.

b. Tell SDS where the dsect is based in the PRV. This is done with the SDS command

`USING` *global global*

for the required dsect *global*.

4. A dsect is generated for each record type which has a name (i.e., is defined in a type declaration). Each field of the record will appear as a variable in this dsect. If `%Test` is 2, the name of the dsect is the type identifier. If `%Test` is 1, the dsect name is generated by the same rule as for external symbols (first four/last four characters of the type name). In this case, if the identifier is longer than 8 characters, the full form will be defined as a label at the beginning of the dsect.

5. A dsect is generated for the local variables of each procedure. This dsect has the name `#`*proc* where *proc* is the procedure name. If `%Test` is 1, it will be shortened to at most 8 characters by taking the first four/last four.

A `USING` command must be given to tell SDS where the dsect is based. This is either R13 at entry to the procedure or R12 after executing the entry sequence.

6. A dsect is generated for the parameter/result area of each procedure. This dsect has the name `!`*proc*, possibly shortened to 8 characters as above. This dsect is based on R1 at entry to the procedure. It may be based on another register at other points.

Only the following SDS data types are used for variables and fields of records:

`F` (with appropriate length) is used for all integer and programmer defined id-list types.

`C` (with appropriate length) is used for fixed *and* varying length strings. Note that for a varying string, the length field is also printed as if it were character, and the variable is printed as the maximum length.

`A` is used for pointers and procedure variables.

`X` is used for anything else.

A duplication factor will be included for arrays. Note, however, that SDS always assumes the lower bound for array subscripts is 1.

V. Using the PDP-11 Plus Compiler

A. Compiler Versions

The current stable version of the PDP-11 Plus compiler is contained in the file `*Plus11`. The file `Plus:Plus11>` contains the latest version for testing. The versions of the Plus-11 compiler are numbered using the same basic scheme as the Plus/370 compiler.

B. Compiling a Program

The compiler is invoked with the `Run` command in a similar way to the 370 compiler. The use of logical units and the `Par=` field are identical, with the exception that unit 0 defaults to the file `Plus:Sourcelib11`.

The output produced by the compiler is equivalent to that produced by the 370 version (with the obvious machine-dependent differences).

C. Running a PDP-11 Program

The PDP-11 version requires that some run-time support routines be provided. These routines are used in the implementation of procedure entry and exit, run-time error checking, and certain operations on string types. These routines are independent of the execution environment of the program.

Other routines are required to set up the stack and global storage and initiate execution of the main program (which should be called `Main`). Another group provide primitive I/O support to the Decwriter console. These routines are system dependent.

At UBC, the object generated by the compiler is combined with the run-time support and any other required code using `*Link11` to generate a binary image which can be loaded into the PDP-11.

1. Use of Link11

The file `*Link11` contains a version of `Link11` which supports pseudo-registers. This supports the commands `PR BEGIN` and `PR END`.

These commands are used to “surround” the inclusion of all Plus modules which are to use the same global area. (The complete input to `Link11` might include independent families of procedures to be linked into one memory image.)

2. Building A Test System

This section describes how a Plus program is currently linked to build a binary image for use with the Test Pdp-11 at UBC. This process will be different at other installations and when building production systems at UBC.

The file `Plus:Objlib11` contains the object for the run-time support routines used. The file `Plus:Freecore` contains a dummy csect which is linked after all code to give the run-time support a handle on the beginning of the “free-core” after all code.

Typical input to `*Link11` to build a test system is therefore

```
SET  @,0200          -- load at 200
LINK PLUS:OBJLIB11
PR BEGIN
LINK Plus_object1
LINK Plus_object2
   ...
PR END
LINK FEP:NEWDEBUG*    -- include debug support
LINK PLUS:FREECORE    -- mark end
SET #,DEBUG           -- enter at DEBUG
MAP map_file
WRITE fep_load_file
STOP
```

`Plus:Objlib11` contains routines required by the object code generated for any Plus program. It also includes object for routines required only for Plus-11 programs compiled with the `"BASIC"` or `"STANDARD"` instruction sets. Other routines implement library subroutines declared in `Plus:Sourcelib11`.

Many applications may wish to select only some of these routines, or to provide substitutes for different system or machine environments. The source, which may be useful as a prototype, is contained in the files `Plus:Lib11*sa` (`*11asr` assembler routines) and `Plus:Lib11*sq` (Plus-11 source routines.)

VI. Using the Motorola 68000 Plus Compiler

A. Compiler Versions

The current stable version of the Motorola 68000 Plus compiler is contained in the file `Plus:Plus68`. The file `Plus:Plus68>` contains the latest version for testing. The versions of the Plus/68000 compiler are numbered using the same basic scheme as the Plus/370 compiler.

B. Compiling a Program

The compiler is invoked with the `Run` command in a similar way to the 370 compiler. The use of logical units and the `Par=` field are identical, with the exception that unit 0 defaults to the file `Plus:Macsourcelib`.

The output produced by the compiler is equivalent to that produced by the 370 version (with the obvious machine-dependent differences).

C. Running a Motorola 68000 Program

Plus/68000 requires that some run-time support routines be provided. These routines are used in the implementation of some operations on string types and long multiplication and division. These routines do not require any global storage. They are supplied in the files `Plus:Obj68MPW`, `Plus:Obj68MDS` and `Plus:Obj68AMI`

The object generated by the compiler is combined with the run-time support and any other required code using the linker corresponding to the `%Target_Operating_System` compiler variable. The first step in this process is to transfer the object file from MTS to the target system, using some binary transmission protocol, such as Kermit. The linking, running and debugging then proceed on the target system.

VII. Source Libraries

A. Library Format

The Plus compilers support a source-library facility which allows segments of source text to be included from library files.

A library consists of a directory followed by 0 or more library members.

The directory consists of 0 or more lines, terminated by either an end-of-file or `/end`. A normal directory record consists of a library member name (which must be a valid Plus identifier) followed by an unsigned integer line number (separated by one or more blanks).

The directory portion may use implicit concatenation (`$Continue with ...` or `$Continue with ... return` records) to specify other libraries to be used. Blank lines and Plus-style comments may also be included within the directory portion.

The line-number in a directory record indicates the line *in the same file* at which the library member begins. The record at the specified location must be `/begin` *membername*, where *membername* is the identifier specified in the directory. A library member ends with an end-of-file or `/end`. Implicit concatenation may be used within a member.

B. Specifying Libraries to the Compiler

Unit 0 on the `Run` command is used to specify the library or libraries to be searched. Multiple libraries may be specified by concatenation either explicitly or implicitly within a library directory.

In effect, the Fdname specified for unit 0 defines the directory to be searched. A library member is *always* obtained from the file in which its defining directory entry is found. If an identifier appears more than once in the libraries to be searched, the first occurrence will be used, without complaint.

C. Including Source From a Library

The `%Include` compiler procedure is used to *conditionally* include library members within a source program.

It appears in the form

`%Include(`id_1`,`id_2`,...,`id_n`)`

An arbitrary number of id_i's may be specified. Each id in the list is considered for inclusion in turn. If the id is not defined at the point where it is considered for inclusion, then the library member with that name is included. If the *id* has been previously defined (as any kind of Plus identifier: type, variable, constant, etc.) then the member is not read in. An error message will be issued if the *id* is not defined and is not in the directory of any library.

`%Include(...)` may be arbitrarily nested within library members.

D. Source Library Utilities

There are two utility programs under ccid Plus that may be of interest to Plus programmers.

1. Plus Library Generator

The program `Plus:Libgen` is a simple program to generate or recreate a Plus library.

It is invoked with an MTS command of the form

```
    Run Plus:Libgen [logical-units] [Par=options]
```

The following logical units may be specified on the `Run` command:

`0`	specifies a file containing an existing Plus source library, or a sequence of library members.
`1`	specifies a file in which a new Plus library is to be built.
`Sercom`	is used to display messages issued by the program.

The `Par` field may specify either or both of the options

`BUILDdir`	indicates that the input from unit 0 has no directory, so one should be built from the information on the `/begin` lines in the input.
`SORTdir`	means that the members in the output library should be sorted alphabetically, rather than preserving the order from the input directory.

(Uppercase letters in the options above indicate allowed abbreviations.)

The input on unit 0 is intended to be an existing Plus library, possibly with extra members that aren't in the directory. The program finds all `/begin` lines in the input file, and uses the names from these to build the output library.

Line numbers in the input file are ignored completely. Comments and blank lines from the input directory, and the order of all members in the input directory, will be preserved in the output library (unless `SORTDIR` is specified).

2. Library Listing Program

The file `Plus:Liblist` contains a simple program that can be used to produce a listing of a Plus source library, with suitable headings etc.

It is invoked with a `Run` command of the form

```
    Run Plus:Liblist [logical-units] [Par=options]
```

The following logical units may be specified:

`Scards`	specifies the library file to be listed.
`Sprint`	specifies a file or device on which the listing is to be produced.

The default output is intended to be suitable for the Xerox 9700 in two-sided, portrait mode. The page numbers and titles are alternated for front/back pages.

The following options may be specified in the `Par` field to modify the output produced. Uppercase letters in the following indicate allowed abbreviations.

`FORMat=`*format-name*	where *format-name* may be one of `LANDSCAPE`, `PORTRAIT`, `UNIVERS_LF`, `TITAN_PF`, or `PLUSLIST`. This specifies the Xerox 9700 format to be used for printing the listing.
`LANDscape`	The listing will be suitable for printing in "landscape" mode, using an output width of 132.

`ONEsided`	The output will be produced for printing onesided. In this case page numbers and titles will not be alternated for front/back pages.
`PAGELENgth=`n	where n is at least 8, specifies the number of lines to be printed per page. The default is 60.
`PAGEWidth=`n	where n is between 76 and 254, specifies the width of the page. The default is 76, which is suitable for “portrait” listings.
`PORTrait`	The output will be suitable for printing in “portrait” mode, using an output width of 76. This is the default.
`SPLit`	If this option is specified, output lines longer than the page width will be split across multiple lines. By default, they are just truncated.
`TWOsided`	The output will be produced for printing twosided. Page numbers and titles alternate between front page and back page formats. This is the default.

VIII. Helpful Hints and Dirty Tricks

This chapter contains a mixed bag of suggestions that should help you to use Plus more efficiently and more effectively. It includes ways of circumventing some of Plus's limitations. These aren't always pretty, but they do work.

Most of these points apply to Plus/370 under MTS, but similar concerns and approaches are often applicable to other environments.

A. Using Equate to Improve Code Generation

When you use an equate statement, the expression being equated to is evaluated once only, at the point where the equate statement occurs. Thus equate is sometimes useful as a way of improving code generation by, in effect, removing common subexpressions. For example, to interchange two elements of an array, something of the following form can be used:

```
equate Source to Arr(I),
   Dest to Arr(I+1);
variable Temp is ... in register;
Temp := Dest;
Dest := Source;
Source := Temp
```

Each of the two subscript calculations has to be performed only once, instead of twice.

In this case, the saving is relatively small, and using the equate may make the program a bit harder to read, so it might not really be an improvement unless the efficiency of these statements was critical. However, if a complex expression is used many times in a procedure, the performance improvement could be substantial, and the use of equate might even make reading the program easier.

A somewhat obscure special case of this is to improve the code generation required to access a reference parameter. If a procedure has a parameter `Par` which is passed by reference, every use of `Par` in the procedure is implicitly an expression dereferencing a pointer; hence a seemingly useless equate like

```
equate Par# to Par
```

(with `Par#` used through the rest of the procedure in place of `Par`), can actually improve the code generation by eliminating this common expression. This is probably only worth doing if performance is critical and the parameter is referenced a lot of times.

You shouldn't try to use equate in this way too much, however, because it uses up registers and may cause worse code to be generated elsewhere, as described in the next section.

B. Plus/370 Register Use

The compiler allocates the general registers for many purposes. Some registers are allocated permanently throughout a procedure, some have a fairly long-term use (across many statements) and some are used during expression calculations. If a procedure needs more registers than are available, the compiler will generate "preemption" code to save and restore registers so the same register can be used for more than one purpose. If this happens a lot, the quality of the code may suffer considerably. Thus it's a good idea to gain some understanding of how the source code for your program affects the register allocation.

One register is allocated for each page of object code and for each page of "stack frame" (local variables, temporaries, etc.) used by the procedure. Up to three registers may be used for each. These registers are allocated for the entire procedure. By keeping procedures small and avoiding using the stack for large variables, you can reduce the number of registers committed for these purposes.

If the procedure has any "storage" parameters, one register is allocated to hold the pointer to the parameter list. This is allocated for the entire procedure.

Up to four registers may be allocated for addressing the most-often referenced global blocks. This number will be reduced if the procedure has more than one code or stack base register, to avoid crowding the rest of the register allocation too much. These registers are allocated at the beginning of the procedure and remain allocated until the last reference to a variable in the global block. For global blocks whose addresses aren't preloaded in the entry code, extra instructions are needed at each reference. If you group your global variable declarations so that each procedure references only a fairly small number of global blocks, the generated code will usually be better. Note that each global variable that isn't in a global block acts as if it were in a global block by itself and so requires separate addressability.

Each open statement and equate statement that involves any expression calculation (including the "implicit" expression involved in using a reference parameter) will require one register to hold the result of the address calculation. (Opens and equates of simple variables don't use up registers, since the resulting variables can be addressed from the same base as the original variable.) These registers are in use from the point of the open or equate statement through to the last reference to the identifier or record fields defined by the statement. Similarly, each register variable requires a register from the point of the declaration through to the point where it is last used.

Overcommitment of registers to equates, opens, and register variables is the most common cause of the compiler generating large numbers of register preemptions. You can minimize the problem by making all such statements as local as possible. That is, don't just put them at the beginning of each procedure, but move them as close as possible to the point where they are really needed.

If compiling a procedure resulted in any register preemptions, a message is printed at the end saying how many were required. Each preemption means one store instruction and *at least* one load instruction. From this you can make some guesses at how the register preemptions have affected code generation for the procedure. Note that a smallish number of preemptions isn't necessarily bad—the performance gains from using register variables, opens, and equates can often be much greater than the losses from any extra preemptions that might result.

C. Execution-time Array Dimensions

Plus does not have any built-in way to define an array whose size is determined at execution time. In practice, however, it is possible to cheat by declaring an array type whose dimensions are the largest that might be required, then defining a pointer variable which points to the array type. The system storage allocation subroutines can then be used to allocate storage for an array of any required size and store its address in the pointer. All references to the array must then be indirectly through the pointer.

For example

```
type Dynamic_Array_Type is array(1 to 9999999) of Integer;
variable Array_Base is pointer to Dynamic_Array_Type;
```

```
...
/* Allocate array of N integers: */
Array_Base := Getspace(0, N * Byte_Size(Integer));
...
/* Initialize the array: */
do I := 1 to N
   Array_Base@(I) := 0
end
```

Note that the compiler will be unable to do any useful subscript checking when an array is defined and allocated in this way, since it believes that any number from 1 to 9999999 is a valid subscript.

In this example, `Byte_Size` is used to determine the size of the array element, which is multiplied by the number of elements required to determine the number of bytes to allocate. Some care is needed when using `Byte_Size` in this way, since the size may not include any slack bytes required by alignment considerations if it is allocated as part of an array. To be absolutely safe, the size of each element of an array of elements of type *t* could be computed as:

```
/* Dummy array to get element size: */
type T1 is array(1 to 2) of t;
constant Element_Size is Byte_Size(T1) - Byte_Size(t)
```

D. Checking For Optional Parameters

Plus doesn't currently provide any built-in way that a procedure can determine whether an optional parameter was supplied by the caller. However, when there are any optional parameters in the declaration, and the last one supplied by the caller is passed by reference, Plus *does* flag it in the high bit as required by the S Type linkage conventions. With a little ingenuity and a lot of cheating it is possible to test for this flag.

The easiest way to accomplish this is to define the parameters as `name` parameters rather than `reference` parameters in the procedure declaration. This doesn't make any difference to the caller, but means that the called procedure can access the pointer passed directly; that is, the implicit dereference is suppressed. (Which means you must *explicitly* dereference it when you want to access that parameter passed.)

You can then equate to the pointer in order to test the high-order bit. For example, given a declaration like

```
procedure Example is
      procedure
      name parameter P1 is ...
      optional name parameter P2 is ...
      end
```

to determine if the caller provided the second parameter, you can test the high-order bit of the pointer to the first:

```
equate Test_Bit to P1 as packed Boolean;
...
if Test_Bit
then
   /* P1 is last parameter so P2 wasn't supplied. */
   ...
else
   /* P2 isn't last parameter. */
   ...
end if
```

If all parameters are optional, there is no way to detect the situation in which the caller provided none. (This isn't supported by the S-Type linkage.)

More generally, if there are a number of optional parameters and you need to determine which was the last one supplied, you can equate an array to the parameter list in order to step through the pointers:

```
procedure Example2 is
      procedure
      name parameter P1 is ...
      optional name parameters P2, P3, P4, P5, P6 are ...
      end;
...
definition Example2
   ...
   equate Pararray to P1 as array (1 to 6) of
         record
            V_Bit is packed Boolean,
            Rest is packed bit(31)
         end;
   ...
   do Number_Of_Parameters := 1 to 6
      exit when Pararray(Number_Of_Parameters).V_Bit
   end;
   /* At this point Number_Of_Parameters specifies
      the total number provided. */
   ...
end Example_2
```

E. Checking Addresses

In Assembler programs under MTS the `BPI` instruction[1] is often used following a reference to a "questionable" address to catch the program interrupt that will result if the address is invalid. This is much more convenient than setting up a program interrupt exit to field such problems.

There is no direct way to do this in Plus. However, `*Plus.Sourcelib` contains a pair of

[1] `BPI` is not a real 370 machine instruction but is simulated by the MTS supervisor.

macros, Fetch_Check and Store_Check which use Inline to test whether the locations referenced by a pointer can be fetched or stored into without a program interrupt occurring.

For example,

```
    Fetch_Check(Ptr, Fetch_Ok)
```

will set Fetch_Ok to True if Ptr@ can be referenced and to False if referencing it causes a program interrupt. Store_Check similarly checks if it can be stored into.

These macros can only be used if the type of Ptr@ has a size of less than 256 bytes. They will check the entire object can be fetched or stored, not just the first byte.

Since the macros inline a BPI instruction, they will only function for programs running under MTS.

To see how the macros handle the Inline'd branch, look at the source in *Plus.Sourcelib.

F. Moving Arbitrary Data

Sometimes it may be necessary to move a specified number of bytes from one memory location to another. If it isn't convenient to use normal Plus types and assignment statements, the easiest way to do this in Plus is to "type cheat" the locations as string variables and use assignment of a Substring.

For example, to move N bytes from the location specified by pointer Source to the location specified by pointer Dest, whatever the types of Source and Dest, you can use something of the form:

```
    equate S to Source@ as character(Maximum_Address),
       D to Dest@ as character(Maximum_Address);

    D := Substring(S, 0, N)
```

(where Maximum_Address is defined by the *Plus.Sourcelib member Machine_Storage_Types).

G. Pointer Arithmetic

It is easy to add or subtract from a pointer by "type cheating" the pointer as an integer, and then operating on the integer.

This is most tidily done by hiding it inside a macro. For example, a macro to add an arbitrary numeric value to an arbitrary pointer is:

```
    macro Increment_Pointer
       parameters are Ptr, Incr;
       equate Cheat_Ptr to Ptr as (0 to Maximum_Address);
       Cheat_Ptr +:= Incr
    end macro
```

A useful variation is a macro to increment a pointer by the size of the item it points to:

```
macro Increment_By_Size
   parameter is Ptr;
   equate Cheat_Ptr to Ptr as (0 to Maximum_Address);
   Cheat_Ptr +:= Byte_Size(Ptr@)
end macro
```

H. Return Codes from Plus Procedures

There isn't currently a built-in way for a procedure written in Plus to return a Type I linkage return code. However, there is a procedure `Return_Code` in `*Plus.Sourcelib` which can be used to fake it.

If a procedure contains

`Return_Code(`*value*`)`

then when the procedure returns, *value* will be passed back as the return code. (The procedure accomplishes this by storing the value in the R15 location in the savearea.)

I. Multilevel Procedure Returns

When a procedure detects an error, it is sometimes useful for it to be able to force a return through more than one level of procedure call. This avoids the necessity of passing back error indications and testing them at all levels.

There are procedures in `*Plus.Sourcelib` that implement a simple form of multilevel return. To use them, the procedure that is to be returned *from* must call `Setup_Return_From` to save necessary information, and the procedure forcing the return calls `Return_From` to effect it. The state information needed is saved in a variable of type `Return_Control_Block_Type`. This must be accessible to both procedures, so global storage is usually used (although it could be passed down as a parameter).

For example:

```
%Include(Return_Control_Block_Type, Setup_Return_From, Return_From);
global Foo

   variable Rcb is Return_Control_Block_Type;
   ...
end Foo;

procedure Level1;

procedure Level2 is
      procedure
      result Success is Boolean
      end;

procedure Level3;

procedure Level4;
```

```
definition Level1
   ...
   if Level2()
   then
      /* It worked OK */
   else
      /* Some error occurred. */
   end if;
   ...
end Level1;

definition Level2
   /* Returns with False is anything goes wrong. */
   Setup_Return_From(Rcb, Success);
   ...
   Level3();
   ...
   return with True
end Level2;

definition Level3
   ...
   if ...
   then
      /* Something wrong.  Return all the way. */
      Return_From(Rcb, False)
   end if;
   ...
   Level4();
   ...
end Level3;

definition Level4
   ...
   if ...
   then
      /* Something wrong.  Return all the way. */
      Return_From(Rcb, False)
   end if;
end Level4
```

In this example, `Level2` sets things up so that any of the procedures it calls (or any procedures called from procedures it calls...) can cause a return as if `Level2` has returned itself. The call to `Setup_Return_From` specifies a variable which is to be set to a return value. Usually, this will be the result value of the procedure calling `Setup_Return_From`. Each of the calls to `Return_From` specify a value to be assigned to this variable before the return occurs. Thus when `Level3` or `Level4` executes the `Return_From`, the effect will be as if `Level2` has assigned the second parameter to `Success` and then returned.

A call to `Return_From` is only valid as long as the procedure which called `Setup_Return_From` is still active; i.e., it hasn't yet returned itself. It *is* possible to call `Return_From` from another routine which gains control asynchronously, such as an attention interrupt routine. In this situation, considerable care is needed to ensure that the `Return_From` is not attempted after the setup routine has returned.

J. Special Linkage Routines

The best advice on special linkage routines is "don't write them if you can possibly avoid it". The process is very tedious and error-prone.

There are a number of predefined linkage routines in the resident Plus library which should handle many of the more common situations requiring special linkage. Some documentation for the existing routines can be found in the `*Forum` conference "Plus". For more information, examine the source in the file `Plus:Newccs2l>sq`.[2]

Special linkage routines can be written in Plus by using `linkage none`. That is, in the following example, the routine `Special_Linkage` contains the entry code required to enter routine `Special`.

```
procedure Special;

procedure Special_Linkage external "SPECLINK";

definition Special
   linkage "SPECLINK";
   /* This routine requires special entry/exit code.  It is
      performed by Special_Linkage. */
   ...
end Special;

definition Special_Linkage
   linkage none;
   /* Linkage routine used to enter routine Special. */
   ...
end Special_Linkage
```

If you must write your own linkage routines, there are a number of macros in `*Plus.Sourcelib` member `Linkage_Macros` which may make it a bit easier. The source for the standard linkage routines, in `Plus:Newccs2l>sq` may also be useful as a model.

[2] This contains versions for use with `%Linkage="NEW"`

APPENDIX A - Implementation Notes and Current Status

The compilers have the same overall organization. They are multi-pass compilers.

The first pass performs all declaration processing, storage allocation and type checking and compiles a tree-representation of the object code. The tree contains a representation of all code-generation semantic actions required. Pass 1 also builds tables describing the variables used by the program.

The next pass tours the tree and produces a stream of pseudo-code. This is very close to the actual machine code that will be produced, but has not yet bound any register usage or determined actual branch addresses. It assumes a slightly idealized machine instruction set.

The register usage is next examined to combine registers where possible. Following this, register allocation is performed.

The pseudo-code is then translated to object machine code, and the object module is written.

At least the following language features are not implemented currently in any compiler:

Anything to do with sets.

Any operations involving reals.

The following additional restrictions of the 370 compiler should also be noted:

The local storage of any procedure may not be bigger than three pages.

The code for any procedure may not be bigger than three pages.

The following additional restrictions apply to the PDP-11 version.

Total size of all procedures must not be bigger than 64K bytes.

Total size of the global pseudo-register area must not be bigger than 64K bytes.

`packed` is only implemented for fields of records.

Variables and constants of type `real` are not implemented.

The following are the additional restrictions of the 68000 compiler:

The local storage of any procedure may not be bigger than 32767 bytes.

The code for any procedure may not be bigger than 32767 bytes.

For the Macintosh, the entire global data area may not be bigger than 32767 bytes. (For the MPW linker, this includes all `entry` constants and constants which contain pointers.)

For the MDS linker, `entry` constants and constants which contain pointers are not implemented.

Variables and constants of type `real` are not implemented.

APPENDIX B - BNF Syntax

The grammar that follows is a slightly simplified version of the LALR(1) grammar used by the compiler. The actual grammar contains rules required by the compiler to perform semantic actions at the appropriate points, rules used by the paragrapher to generate paragraphed listings, and additional rules to make the language accept redundant semicolons and commas in a variety of contexts.

```
<program> ::= <statement_list> end_of_file

<statement_list> ::= <statement>
                   | <statement_list> ; <statement>

<statement> ::= <type_declaration>
              | <variable_declaration>
              | <constant_declaration>
              | <procedure_declaration>
              | <macro_declaration>
              | <open_declaration>
              | <equate_declaration>
              | <escape>
              | <return>
              | <if_statement>
              | <%if_statement>
              | <select_statement>
              | <assertion>
              | <assignment>
              | <compound>
              | <storage_reference>
              | <procedure_definition>
              | <global_pack>

<global_pack> ::= global identifier <external_name> <statement_list> <global_end>

<global_end> ::= end <optional_id>
               | end global <optional_id>

<optional_id> ::= identifier
                | empty

<external_name> ::= external <constant_expression>
                  | empty

<type_declaration> ::= type <id_list> is <type>
                     | <type_declaration> , <id_list> is <type>

<variable_declaration> ::= variable <declaration_element>
                         | <variable_declaration> , <declaration_element>

<declaration_element> ::= <id_list> is <type> <allocation>
```

```
<allocation> ::= in register
             | in register <constant_expression>
             | in register <constant_expression> to <constant_expression>
             | in storage
             | at absolute <constant_expression>
             | external
             | external <constant_expression>
             | entry
             | entry <constant_expression>
             | empty

<constant_declaration> ::= constant <id_list> is <constant_expression> <allocation>
                       | <constant_declaration> , <id_list> is <constant_expression> <allocation>

<macro_declaration> ::= <macro_head> <macro_body> <macro_end>

<macro_end> ::= end <optional_id>
              | end macro <optional_id>

<macro_head> ::= macro identifier <macro_parameters> ;

<macro_parameters> ::= parameter is <id_list>
                   | empty

<macro_body> ::= <statement_list>
               | <parenthesized_expression>

<open_declaration> ::= open <open_element>
                   | <open_declaration> , <open_element>

<open_element> ::= <storage_reference> <equate_type>

<equate_declaration> ::= equate <equate_element>
                     | <equate_declaration> , <equate_element>

<equate_element> ::= identifier to <storage_reference> <equate_type>

<equate_type> ::= as <type>
               | empty

<procedure_declaration> ::= <procedure_head> <id_list> <proc_specifications>
                        | <procedure_decl>

<procedure_decl> ::= procedure <id_list> is <type> <proc_specifications>
                 | <procedure_decl> , <id_list> is <type> <proc_specifications>

<proc_specifications> ::= <proc_specifications> external <constant_expression>
                      | <proc_specifications> <linkage>
                      | <proc_specifications> stacksize <constant_expression>
                      | empty
```

```
<linkage> ::= linkage <constant_expression>
            | linkage system
            | linkage none

<procedure_definition> ::= <definition_head> <statement_list> <procedure_end>

<definition_head> ::= definition identifier
                    | definition identifier <entry_options> ;
                    | <procedure_declaration> definition
                    | <procedure_declaration> definition <entry_options> ;

<entry_options> ::= <entry_option>
                  | <entry_options> <entry_option>

<entry_option> ::= <linkage>
                 | environment <storage_reference>

<procedure_end> ::= end <optional_id>
                  | end procedure <optional_id>
                  | end definition <optional_id>

<id_list> ::= identifier
            | <id_list> , identifier

<type> ::= <attribute> <type>
         | <basic_type>

<basic_type> ::= ( <id_list> )
               | bit ( <constant_expression> )
               | real ( <constant_expression> )
               | character ( <constant_expression> )
               | character ( <constant_expression> to <constant_expression> )
               | ( <constant_expression> to <constant_expression> )
               | <record_type>
               | pointer to <type>
               | set of <type>
               | array <type> of <type>
               | <procedure_type>
               | unknown
               | global ( <constant_expression> )
               | identifier

<attribute> ::= packed
              | aligned <alignment> left
              | aligned <alignment> right
              | fast
              | small
              | value
              | left
              | right
```

```
             | environment <type>
             | system

<alignment> ::= <constant_expression>
             | <constant_expression> in <constant_expression>

<record_type> ::= record <field_list> <variant_part> <end_record>

<end_record> ::= end
              | end record

<field_list> ::= <declaration_element>
             | <field_list> , <declaration_element>
             | empty

<variant_part> ::= <variant_list>
                | <variant_list> <variant_else>
                | empty

<variant_list> ::= <variant_list> <variant_element>
               | <variant_head>

<variant_head> ::= variant identifier of <type> from
                | variant <type> from

<variant_element> ::= <variant_label_list> :  <field_list>

<variant_label_list> ::= case <constant_expression>
                     | <variant_label_list> , <expression>

<variant_else> ::= else <field_list>

<procedure_type> ::= procedure <parameter_list> <result_part> end

<parameter_list> ::= <parameter_list> <parameter_part>
                  | empty

<parameter_part> ::= <parameter_kind> <declaration_element>
                  | <parameter_part> , <declaration_element>

<parameter_kind> ::= <optional> <reference> parameter

<optional> ::= optional
            | repeated
            | empty

<reference> ::= name
             | reference
             | empty
```

```
<result_part> ::= result <declaration_element>
                | optional result <declaration_element>
                | empty

<escape> ::= <escape_type> <optional_label> <when_unless>

<escape_type> ::= exit
                | repeat

<optional_label> ::= <label>
                   | empty

<label> ::= < identifier >

<when_unless> ::= when <expression>
                | unless <expression>
                | empty

<return> ::= return <when_unless> <with_part>
           | return <with_part> <when_unless>
           | return <when_unless>

<with_part> ::= with <expression>

<if_statement> ::= if <if_then_else> <end_if>

<if_then_else> ::= <expression> <then_part> <else_part>

<end_if> ::= end
           | end if

<then_part> ::= then <statement_list>

<else_part> ::= else <statement_list>
              | elseif <if_then_else>
              | empty

<%if_statement> ::= %if <%if_then_else> <%end_if>

<%if_then_else> ::= <constant_expression> <%then_part> <%else_part>

<%end_if> ::= %end
            | %end %if
            | %end if

<%then_part> ::= %then <statement_list>

<%else_part> ::= %else <statement_list>
               | %elseif <%if_then_else>
               | empty
```

```
<select_statement> ::= <select_start> <select_alternatives> <end_select>

<select_start> ::= select <expression> from

<end_select> ::= end
             | end select

<select_alternatives> ::= <select_alternatives_list>
                      | <select_alternatives_list> else <statement_list>

<select_alternatives_list> ::= <select_alternatives_list> <select_alternative>
                           | empty

<select_alternative> ::= <select_label_list> :  <statement_list>

<select_label_list> ::= case <constant_expression>
                    | <select_label_list> , <expression>

<assertion> ::= assert <expression>

<assignment> ::= <storage_reference> <assign_op> <expression>
             | <storage_reference> , <assignment>

<assign_op> ::= :=
            | <adding_op> :=
            | <multiplying_op> :=

<compound> ::= <label> <unlabelled_compound> <label>
           | <unlabelled_compound>

<unlabelled_compound> ::= begin <statement_list> end
                      | cycle <statement_list> <end_cycle>
                      | <do_head> <statement_list> <end_do>

<end_cycle> ::= end
            | end cycle

<end_do> ::= end
         | end do

<do_head> ::= do <storage_reference> := <expression> <direction> <expression>
          | do <storage_reference> := each <expression>

<direction> ::= to
            | downto

<storage_reference> ::= identifier
                    | % identifier
                    | <procedure_or_array_reference> )
                    | <storage_reference> ( <return_code> )
                    | <storage_reference> . identifier
                    | <storage_reference> @
```

```
<procedure_or_array_reference> ::= <subscripted_reference>
                                 | <subscripted_reference_head>
                                 | <subscripted_with_return_code>

<subscripted_reference> ::= <subscripted_reference_head> <expression>

<subscripted_reference_head> ::= <storage_reference> (
                               | <subscripted_reference> ,

<subscripted_with_return_code> ::= <with_return_code_head> return code <storage_reference>
                                 | <with_return_code_head> with <storage_reference>

<with_return_code_head> ::= <subscripted_reference>
                          | <subscripted_reference_head>
                          | <subscripted_with_return_code> ,
                          | <subscripted_with_return_code>

<constant_expression> ::= <expression>

<expression> ::= <logical_formula>
               | <conjunction>
               | <disjunction>

<disjunction> ::= <logical_formula> or <logical_formula>
                | <disjunction> or <logical_formula>

<conjunction> ::= <logical_formula> and <logical_formula>
                | <conjunction> and <logical_formula>

<logical_formula> ::= <arithmetic_expression> <relation> <arithmetic_expression>
                    | <arithmetic_expression>

<relation> ::= <relation_op>
             | ¬ <relation_op>
             | not <relation_op>

<relation_op> ::= <
                | < =
                | < <
                | < < =
                | >
                | > =
                | > >
                | > > =
                | =
                | in
                | subset

<arithmetic_expression> ::= <term>
                          | <arithmetic_expression> <adding_op> <term>
```

```
<adding_op> ::= +
              | -
              | ||
              | |
              | xor

<term> ::= <primary>
         | <term> <multiplying_op> <primary>

<multiplying_op> ::= *
                   | /
                   | mod
                   | &

<primary> ::= <unary_op> <primary>
            | <storage_reference>
            | number
            | bit_string
            | string
            | <parenthesized_expression>
            | <set> }
            | { }

<parenthesized_expression> ::= ( <expression> )

<set> ::= { <expression>
        | <set> , <expression>

<unary_op> ::= +
             | -
             | ¬
             | not
             | abs
```

APPENDIX C - Plus Reserved Words

The following keywords are reserved in the current versions of Plus, and cannot be used as programmer-defined identifiers.

`abs`	`absolute`	`aligned`	`and`
`are`	`array`	`as`	`assert`
`at`	`begin`	`bit`	`case`
`character`	`code`	`constant`	`constants`
`cycle`	`definition`	`do`	`downto`
`each`	`else`	`elseif`	`end`
`entry`	`environment`	`equate`	`exit`
`external`	`fast`	`from`	`global`
`if`	`in`	`is`	`left`
`linkage`	`macro`	`mod`	`name`
`none`	`not`	`of`	`open`
`optional`	`or`	`packed`	`parameter`
`parameters`	`pointer`	`procedure`	`procedures`
`real`	`record`	`reference`	`register`
`registers`	`repeat`	`repeated`	`result`
`return`	`right`	`select`	`small`
`stacksize`	`storage`	`subset`	`system`
`then`	`to`	`type`	`types`
`unknown`	`unless`	`use`	`value`
`variable`	`variables`	`variant`	`when`
`with`	`xor`		

The following words will be reserved in a future version and should also be avoided.

`descriptor`	`set`	`signed`	`unsigned`

APPENDIX D - Plus/370 Linkage Conventions and Run-time Organization

This appendix describes linkage conventions used by Plus/370 and the associated run-time organization required.

A. Register Usage

The entry sequence of a Plus procedure loads the stack bases and then the code bases. These are allocated from register 12 down, skipping register 11, using as many registers as required. Register 11 contains the address of the global environment, normally set up before entry. Thus if a procedure requires one stack base and one code base (the normal case), register 12 will be the stack base and register 10 the code base.

R13 is loaded with the "next stack frame" as part of the entry sequence, if the procedure contains any calls to other procedures.

Other global addressability may be set up using registers below those needed for the code bases.

B. Stack and Global Organization

The old version of the Coding Conventions linkage (`%Linkage="OLD"` or `%Linkage="STANDARD"`) places no requirements on the organization of the stack and global storage. At entry to a Plus procedure, R11 should contain the address of the global storage if the procedure uses any globals, and R13 should contain the address of an area that can be used as a stack.

These requirements are also present with `%Linkage="NEW"`, but there are additional requirements on the way global storage and the stack must be initialized at the time it is allocated.

The bottom six words of the space allocated as a stack must now be initialized to contain a **stack descriptor**. The stack descriptor contains pointers to the first word of the stack space, the last word of the stack space (used for checking for stack overflow), and a word in which the current top-of-stack is saved if a call is done to a procedure with the `system` attribute. It also contains forward and backward links to other stacks that may be used for attention and program interrupt routines etc. The exact format of this stack descriptor is described by `Long_Stack_Descriptor_Type` in the library `*Plus.Sourcelib`.

It is also required that R11 be set to point to a global storage area, whether or not the procedure actually uses global variables. The first two words of global storage are now reserved. The first is reserved for a pointer to the "CLS transfer vector" for internal system use. This should normally be initialized to the value `Address(Stdtv)`. The second word must contain a pointer to a **short stack descriptor**. The short stack descriptor is a four word area that contains a copy of the first three words from the stack being used and a pointer to the long stack descriptor. Its format is described by `Short_Stack_Descriptor_Type` in `*Plus.Sourcelib`. The LCSPR's `QLCSPR` and `QGLOBAL` both define symbols `CLSTVPTR` and `STK_DESC` for these two words.

The short stack descriptor is used to provide a level of indirection in retrieving the stack from global storage. An application will only ever allocate one short stack descriptor. If the application is using more than one global storage environment, *all* environments must point to the same short stack descriptor, which will in turn point to the stack currently in use.

When initializing a stack descriptor, the stack end pointer should actually be set to 72 bytes before the true end, to allow room for registers to be saved in either a coding conventions or OS linkage entry sequence before any limit checking is done.

Note that when the environment is defined as `pointer to ...`, the specified record type *must* conform to the above requirements. That is, the first two words of the record must be reserved for the pointer to the CLS transfer vector and the pointer to the short stack descriptor, and must be appropriately initialized.

C. Plus Procedure Linkage

The normal procedure linkage used is the MTS coding conventions linkage, except that all (non-result) registers are restored on return from a call (and are presumed to be restored by any procedures called).[1]

The linkage code consists of:

1. Prelude (performed by caller)

Load register parameters, and assign storage parameters.

Load R1 with address of storage parameters/result if any.

For `linkage none` routines only, load R13 with the next stack frame address (in all other cases it is loaded in caller's entry sequence).

If called procedure has the `system` attribute, and if `%Linkage="NEW"`, update the stack descriptor to indicate the top-of-stack at the point of the call.

Load R15 with address of called procedure.

If the call specifies `with ...`, load R11 with the specified environment.

Call procedure via

```
BALR R14,R15
```

2. Entry (performed by called procedure)

a. normal Plus linkage:

Save all 16 general registers on the stack via

```
STM R0,R15,0(R13)
```

Save any register parameters as local variables on the stack.

Load code base(s).

Load stack base(s).

If `%Linkage="NEW"` and `%Stack_Check` is true, perform stack overflow check.

If the procedure contains any calls to other procedures, load the next stack frame.

If the procedure has storage parameters, copy R1 to another register.

If the definition specifies the `environment` option, load R11 with the new environment.

[1] The new (1986) version of the coding conventions also assume this.

Load “permanently assigned” global base registers.

Note the register parameters are re-saved separately from the entry save-area, so that the caller can assume all non-result registers are restored.

Register results are allocated in the save-area so that no special action is required by the return.

b. `linkage "`*extname*`"`:

For a “special linkage” routine, in place of the first step (storing all registers on the stack), the compiler generates code which transfers to the external symbol *extname* from the entry sequence of the routine being entered (`SUB1` in the following).

The exact sequence generated is

```
SUB1     L    R15,LINKADDR      Get address of linkage code
         BALR R15,R15           And go there
         BALR R15,0             Reestablish addressability
         B    SUB1CONT          Skip "parameters"
         DC   F'stacksize'
         DC   F'framesize'
         CXD
LINKADDR DC   V(extname)        Address of linkage code
SELFADDR DC   V(SUB1)           Address of routine being entered
GLOBALID DC   XL4'global-id'    Global id
STAKOFF  DC   Q(STK_DESC)       Offset of stack descriptor
SUB1CONT L    R10,SELFADDR      Set code addressability
```

Thus the linkage code is entered with R15 containing the return address, and with a number of “parameters” accessible via offsets from R15.

The requested stacksize is at 6(R15). If this was not specified in the procedure declaration, then it will be compiled as a weak external reference to the symbol `STAKSIZE`. This allows specifying the stack size at load time by including an absolute DEF loader record in the object deck.

The word at 10(R15) contains the actual stack space requirements of `SUB1`, which may be of interest to some linkage routines.

The word at 14(R15) is the total global area size of the loaded program containing SUB1. (Note that in general, a linkage routine should use this as the global size, rather than including a CXD or use of `Global_Size` within the linkage code itself. This is because under some circumstances, is is possible that the linkage routines might be part of an earlier load.) If the procedure's environment is specified as `pointer to` *record-type*, this word will contain the size of the record.

If the procedure's environment is of type `global(`*global-id*`)`, the word at 18(R15) contains the value of *global-id*. Otherwise this word is 0. This value may be useful as a code to the `Gpsect` subroutine for allocating or retrieving the global storage.

The word at 22(R15) contains the offset of `STK_DESC` within the pseudo-register vector. When `%Linkage="NEW"`, this value *must* be 4. It is just used for error-checking purposes by linkage routines to test whether the required loader records (see Chapter IV, page 127) were present when the program was loaded.

The linkage code is required to set the address of a stack in R13 (*not* `Stack_Base_Register`). The register values to be restored at return should be stored out into the bottom of the stack. It should also load R11 (`Global_Base_Register`) with the address of the global environment. (R11 *must* be loaded when `%Linkage="NEW"`.)

The linkage routine returns to the called procedure at the address in R15. If `Sub1` was called with any register parameters, these must be in the original registers on return to `Sub1`, and if it has storage parameters, R1 must be preserved.

The called procedure will then continue the entry sequence as for a normal Plus program, as described above.

When the called procedure (`Sub1`) returns, it will load all 16 registers from the bottom of the stack, then branch on R14. Thus the linkage routine may intercept the return by leaving an appropriate value in the 15th word of the savearea. Ultimately, of course, the linkage routine must ensure that the registers on return to the caller of `Sub1` are consistent with the environment expected by the caller.

The exact steps to be performed in the linkage code will vary depending on the situation. The following are the most common situations (with `%Linkage="NEW"` assumed):

•• If initializing the entire application, allocate one or more stacks and initialize the long stack descriptor at the bottom of each. Allocate the short stack descriptor, and set it up to correspond to the first stack to be used. Allocate the global storage for the routine being actually entered, and make it point to the short stack descriptor.

•• If initializing a subroutine that uses its own global storage, allocate the global storage and make it point to the existing short stack descriptor.

•• If reentering the Plus world from a Fortran-type routine that was previously called from a Plus routine, retrieve the global storage (somehow), and from it, obtain the short stack descriptor, and thence the top of stack at the point that the Plus routine called the Fortran one.

•• If entering a routine asynchronously (e.g., an attention interrupt handler), switch to a new stack. This is done by retrieving the global storage, from it the short stack descriptor, and then the current stack. The short stack descriptor information is then copied to the long stack descriptor for the current stack. The next stack is then obtained, from the links in the current stack, or by allocating and initializing a new one, and the short stack descriptor is reinitialized from the new stack.

In general, on returning from a procedure with special linkage code, the exit code used should undo whatever was done in the entry code. However, in some situations it is more desirable to allocate space the first time (via `Gpsect`), and to not free it on return, so that subsequent calls will be cheaper.

c. `linkage system`:

`linkage system` is implemented by the compiler as if `linkage "QSYSENTR"` had been specified. That is, entry code is as described above, branching to a special entry routine in the resident system.

The following comments assume `%Linkage="NEW"`. The operation of `QSYSENTR` is somewhat different (and much less efficient) for the older linkage.

The linkage routine uses the MTS subroutine `Gpsect` to allocate (the first time) or retrieve (subsequently) the global storage, using the *global-id* from the linkage parameters. It retrieves the stack from global storage, or allocates it the first time by using `Getspace`. The stack size specified will be determined from the linkage parameters. A one page stack will be allocated if 0 is specified.

On return from the `linkage system` routine, the stack and global storage are *not* released.

This linkage code does not set up a program interrupt handler to intercept run-time error conditions within the Plus code.

d. `linkage none`:

For `linkage none`, no entry code is generated. The procedure must use `Inline` and register variables to “bootstrap” to the point where Plus code can execute correctly. The predefined register variables `Code_Base_Register`, `Stack_Base_Register` and `Environment_Base_Register` can be used in setting up the Plus entry requirements.

A great deal of care is required since the compiler will assume the code/stack/global bases etc. have been set up correctly if any statements in the procedure require them. It is advisable to turn code listing on to see if all is as planned. It may also be prudent to set `%Preempt` to false, to prevent the compiler for doing unexpected register preemptions, which might interfere with the expected code.

In particular, note that the `Code_Base_Register` must be loaded with the address of the entry point of the routine before any branches, (including run-time checking), or references to the constant pool. The `Stack_Base_Register` must be loaded with the address of a stack before any instructions requiring temporaries, or referencing local variables. The `Environment_Base_Register` must be loaded before any references to global variables.

A `linkage none` routine returns by doing just a `BR R14`; you must make sure any other required register restoration is done before returning.

The exact details of how all this should be accomplished depend, of course, on the environment from which the routine is being called.

3. **Exit** (performed by called procedure)

Restore all registers:

```
LM R0,R15,0(R12)
```

Return to caller

```
BR R14
```

For `linkage none`, only the `BR` is generated; the registers are not restored.

4. **Postlude** (performed by caller)

Restore R11 to the caller’s environment if it was changed before the call.

Store the return code (R15) if the return-code phrase occurs in the call.

D. Stack Frame Layout

The usage of the stack by the Plus/370 compiler is as follows:

```
           |          . . .                 |  high address
  called   | local variables                |
procedure  | register parameter area        |
           | register savearea (R0–R15)     <—— R13 at call, R12 inside procedure
    ↑      |                                |
    |      | ... (possible temporaries)     |
    |      |                                |
           | result (if any)                |
  calling  | storage parameters             <—— R1 at call points here
procedure  |          . . .                 |
           |                                |  low address
```

APPENDIX E - Plus-11 Linkage Conventions and Run-time Organization

This appendix describes linkage conventions used by Plus-11 and the associated run-time organization required.

A. Object Modules

Plus-11 generates object modules in the form expected by `*Link11`, which is essentially the same as the IBM object module format used by MTS. The code generated depends on auxiliary routines, provided by Plus:Objlib11 to perform procedure entry and exit sequences, certain Plus operations (character handling), and for run-time check processing.

B. Register Usage

By default, the code generated by Plus-11 uses R5 to point to the local stack frame of the current procedure, and R4 points to the base of the pseudo-register (global storage). If the option `%Linkage:="ALTERNATE"` is specified, the use of R4 and R5 is reversed.

C. Parameter Passing

Parameters and results of Plus-11 procedures are normally passed through the stack. Space for the result (if any) is allocated on the stack by the calling procedure, and the values of parameters are then pushed on the stack.

The called procedure accesses parameters and result locations by positive offsets from the local stack frame pointer (R5) and local variables by negative offsets from the frame pointer.

D. Procedure Linkage

Procedure linkage in Plus-11 programs is performed by the following sequence:

1. Prelude (performed by caller)

Adjust SP to leave space for result if any.

Push parameters on the stack.

Call procedure via

`JSR PC,`*procname*

2. Entry (performed by called procedure)

a. normal Plus linkage:

Save old stack frame pointer on the stack.

Save registers R0 - R4 on the stack.

Adjust SP to point above register save area.

Adjust R5 to point to the new stack frame.

Allocate space for local variables by adjusting the stack pointer.

All but the last step is accomplished by a run-time routine PLUSENTR. The first instruction of each Plus-linkage procedure is

```
JSR R5,PLUSENTR
```

b. `linkage "`*extname*`"`:

For a “special linkage” routine, in place of the call to `PLUSENTR`, the compiler generates code to branch to the external symbol *extname* from the entry sequence. The exact sequence generated is:

```
SUB1      JSR   R5,extname         Go to linkage code
          BR    SUB1CONT           Skip "parameters"
          DC    F'stacksize'
          DC    F'framesize'
          DC    F'0'               Reserved for global size someday
SUB1CONT ...
```

The code at *extname* should save the registers on the stack, set up global storage, etc., as required. It should return with R5 set to point to the new stack frame. Following the return from *extname* the entry code will allocate space for local variables as with the normal Plus linkage.

c. `linkage system`:

Plus-11 compiles `linkage system` the same as for the normal Plus linkage.

d. `linkage none`:

For `linkage none`, no entry code is generated. The procedure must use `Inline` and register variables to “bootstrap” to the point where Plus code can execute correctly. The predefined register variables `Program_Counter`, `Stack_Pointer`, `Frame_Base_Register` and `Global_Base_Register` can be used in setting up the Plus entry requirements.

A great deal of care is required since the compiler will assume the code/stack/global bases etc. have been set up correctly if any statements in the procedure require them. It is advisable to turn code listing on to see if all is as planned.

In particular, note that `Frame_Base_Register` must be set up before any references to local variables, and `Global_Base_Register` must be set before any references to global variables.

The exact details of how all this should be accomplished depend on the environment from which the routine is being called.

3. Exit (performed by called procedure)

For all linkage kinds except `linkage none`, a Plus return is just compiled into

```
JMP PLUSEXIT
```

This undoes the entry sequence, leaving SP pointing to the top of the parameter area.

A `linkage none` routine returns by doing just an `RTS PC`; you must make sure any other required register restoration is done before returning.

4. Postlude (performed by calling procedure)

Collapses the space for parameters (if any), leaving SP pointing to the result variable (if there is one).

E. Stack Frame Layout

As a result of the above linkage conventions, the usage of the stack by a Plus11 procedure is as follows. (Note this illustration is the opposite way up from the preceding Plus/370 version, since Plus-11 stacks grow downwards in memory.)

```
            |             ...              |  high address
  calling   | result (if any)              |
 procedure  | parameters (if any)          |
            | old PC (return address)      |
     ↓      | old stack frame (R5)         ←— new R5 points here
            | register save area (R0–R4)   |
  called    | local variables              |
 procedure  |             ...              |
            |                              |  low address
```

Note local variables are accessed by negative offsets from R5, parameters and results are accessed by positive offsets.

APPENDIX F - Plus/68000 Linkage Conventions and Run-time Organization

This appendix describes linkage conventions used by Plus/68000 and the associated run-time organization required. The details of this vary depending on whether the code is generated for the AMIGA or the Macintosh, as specified by the `%Target_Operating_System` compiler variable.

A. Macintosh System Support

The Macintosh system provides a basic application runtime environment with a global data area addressed by register `A5` and a series of independently loaded **segments** of code. The system uses the global data area with positive offsets from `A5`, while the area with negative offsets is for the global storage of the program. This fits fairly well with Plus's notion of global storage, but it does not encourage the switching of environments. For this reason Plus/68000 implements environments which are pointers to record types by using register A4 as the environment base register. One of the system data structures in the positive offset area is the **jump table**. This is used to do procedure calls from one segment to another. Such calls can cause a segment to be implicitly loaded. When a segment is loaded, only the jump table is relocated. *No other relocation is done*. This makes it difficult to implement Plus constants which contain pointers.

1. Macintosh Programmer's Workshop (MPW)

When the `%Target_Operating_System` compiler variable has the value `"MAC/MPW"`, Plus/68000 generates object modules in the form expected by the Macintosh Programmer's Workshop (MPW) Linker. The MPW linker provides for the initialization of the global data area, including pointers which point to other global data areas. Thus with MPW, Plus/68000 uses the global data area for all "entry" constants and for any constants which contain pointers. This does have the drawbacks of requiring all such constant data to be copied from the initialization segment (`%A5_Init`) to the global data area, and enlarging the global data area.

The code generated depends on auxiliary routines, provided by Plus:Obj68MPW, to perform some string operations, longword multiply and divide, and array of packed operations.

2. Macintosh Development System (MDS)

When the `%Target_Operating_System` compiler variable has the value `"MAC/MDS"`, Plus/68000 generates object modules in the form expected by the Macintosh Development System (MDS) Linker. The MDS linker does not provide for the initialization of the global data area. Thus with MDS, Plus/68000 does *not* implement "entry" constants or constants which contain pointers.

The code generated depends on auxiliary routines, provided by Plus:Obj68MDS, to perform some string operations, longword multiply and divide, and array of packed operations.

B. AMIGA System Support

When the `%Target_Operating_System` compiler variable has the value `"AMIGA"`, Plus/68000 generates object modules in the form expected by the AMIGA linkers. The code generated depends on auxiliary routines, provided by Plus:Obj68AMIGA, to perform some string operations, longword multiply and divide, and array of packed operations.

C. Register Usage

The code generated by Plus/68000 for the Macintosh uses register `A5` to point to the global data for the program, and `A6` to point to the local stack frame of the current procedure. For the AMIGA, the use of these two registers is reversed.

D. Parameter Passing

Parameters and results of Plus/68000 procedures are normally passed through the stack. Space for the result (if any) is allocated on the stack by the calling procedure, and the values of parameters are then pushed on the stack.

The called procedure accesses parameters and result locations by positive offsets from the local stack frame pointer and local variables by negative offsets from the frame pointer.

E. Macintosh Procedure Linkage

Procedure linkage in Plus/68000 Macintosh programs is performed by the following sequence:

1. **Prelude** (performed by caller)

Adjust SP to leave space for result, if any.

Push parameters on the stack, in left to right order.

Push space on the stack for any omitted optional parameters.

If the call specifies `with ...`, load `A4` with the specified environment.

Call procedure via

`JSR PC,`*procname*

or, for system procedures, the instruction supplied as the external name of the procedure.

2. **Entry** (performed by called procedure)

If there are any local variables, save the old stack frame pointer on the stack, adjust stack frame pointer to point to this saved value, and adjust stack pointer to allocate space for the local variables via

`link A6,`*local-stack-size*

Save any registers in `D3-D7` or `A2-A5` which are modified by the called procedure, usually with

`movem.l -(SP),`*register-mask*

If the definition specifies the `environment` option, load `A4` with the new environment.

3. **Exit** (performed by called procedure)

Any saved registers are restored, usually with

`movemfm.l (SP)+,`*register-mask*

If there are any local variables, the stack pointer is restored from the stack frame pointer, then the stack frame pointer is restored by popping it off the stack, via

```
unlk A6
```

If there are no parameters, the procedure returns via

```
rts
```

If there are parameters, the called procedure has to pop the saved PC via

```
movea.l (SP)+,A0
```

remove the parameters via

```
addq.l SP,#n
```

or

```
lea n(SP),SP
```

then return by

```
jmp (A0)
```

4. **Postlude** (performed by calling procedure)

   If the call specifies `with ...`, restore the calling procedure's environment base register (`A4`).

   The result, if any, is popped after it has been used.

F. AMIGA Procedure Linkage

Procedure linkage in Plus/68000 AMIGA programs is performed by the following sequence:

1. **Prelude** (performed by caller)

   Adjust SP to leave space for result, if any.

   Push parameters on the stack, in right to left order.

   If the call specifies `with ...`, load `A6` with the specified environment.

   Call procedure via

   ```
   JSR PC,procname
   ```

   or, for system procedures

   ```
   JSR PC,procname(A6)
   ```

2. **Entry** (performed by called procedure)

   If there are any local variables, save the old stack frame pointer on the stack, adjust stack frame pointer to point to this saved value, and adjust stack pointer to allocate space for the local variables via

   ```
   link A6,local-stack-size
   ```

   Save any registers in `D2-D7` or `A2-A4` or `A6` which are modified by the called procedure, usually with

   ```
   movem.l -(SP),register-mask
   ```

   If the definition specifies the `environment` option, load `A5` with the new environment.

3. Exit (performed by called procedure)

Any saved registers are restored, usually with

```
movemfm.l (SP)+,register-mask
```

If there are any local variables, the stack pointer is restored from the stack frame pointer, then the stack frame pointer is restored by popping it off the stack, via

```
unlk A6
```

The procedure returns via

```
rts
```

4. Postlude (performed by calling procedure)

The parameters are popped from the stack.

If the call specifies `with ...`, restore the calling procedure's environment base register (`A6`).

The result, if any, is popped after it has been used.

G. Stack Frame Layout

As a result of the above linkage conventions, the usage of the stack by a Plus/68000 procedure is as follows. (Note this illustration is the opposite way up from the Plus/370 version, since Plus/68000 stacks grow downwards in memory.)

	. . .	high address
calling	result (if any)	
procedure	parameters (if any)	
	old PC (return address)	
↓	old stack frame base	⟵ new stack frame base points here
	local variables	
called	register save area	
procedure	. . .	
		low address

Note local variables are accessed by negative offsets from the stack frame base, parameters and results are accessed by positive offsets.

APPENDIX G - Plus/68000 `Inline` Codes

This appendix gives the strings that Plus/68000 recognizes for the first operand of `inline`.

The recognized opcodes are:

abcd	add	addm	adda	addi
addq	addx	and	andm	andi
andicc	andis	asl	asr	aslm
asrm	atrap	bCC	bchgd	bchg
bclrd	bclr	bsetd	bset	bsr
btstd	btst	chk	clr	cmp
cmpa	cmpi	cmpm	dbCC	divs
divu	eorm	eori	eoricc	eoris
exg	ext	illegal	jmp	jsr
lea	link	lsl	lsr	lslm
lsrm	move	movefcc	movecc	movesr
movefsr	moveusp	movefusp	movea	movec
movefc	movem	movemfm	movep	movepfm
moveq	moves	muls	mulu	nbcd
neg	negx	nop	not	or
orm	ori	oricc	oris	pea
reset	rol	ror	rolm	rorm
roxl	roxr	roxlm	roxrm	rtd
rte	rtr	rts	sbcd	sCC
stop	sub	subm	suba	subi
subq	subx	swap	tas	trap
trapv	tst	unlk		

The codes for the size part are the usual `b` for byte, `w` for word (two bytes), and `l` for long (four bytes).

Those opcodes above that end in `CC` are formed by replacing the `CC` with one of the following condition codes:

t	f	hi	ls	hs
lo	ne	eq	vc	vs
pl	mi	ge	lt	gt
le				

As well, `cc` is accepted for `hs`, `cs` for `lo`, `bra` for `bt` and `dbra` for `dbf`.

The addressing mode specifications are:

dr	ar	(ar)	(ar)+	-(ar)
d(ar)	d(ar,xr)	abs.w	abs.l	d(pc)
d(pc,xr)	#	=		

The `=` mode is used to indicate a PC relative reference to a constant in the literal pool.