THE UNIVERSITY OF MANITOBA

FLEX: A FORTRAN Language Extension

by

Marian E. Power

A thesis

submitted to the Faculty of Graduate Studies

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

WINNIPEG, MANITOBA

October 1974

FLEX: A FORTRAN LANGUAGE EXTENSION

by

Marian E. Power

A dissertation submitted to the Faculty of Graduate Studies of the University of Manitoba in partial fulfillment of the requirements of the degree of

MASTER OF SCIENCE

Abstract

The programming language FLEX is an extension of FORTRAN which allows the definition of new data types and new operators, as well as the extension of present operators to include the new data types. It incorporates the four most useful control structures - DO and WHILE loops, CASE and IF selections - which are similar to the corresponding constructs of PL/1 and ALGOL. FLEX also makes other improvements to FORTRAN, such as stream input (as in PL/1) and embedded format specifications in input/output statements.

The ability to define new data types permits the user to define ones which are appropriate for his application, rather than simply using existing data types. He can define new operations on precisely those data types he requires. Thus he is much more specific in the declaration and use of variables than in either FORTRAN or PL/1.

Expressions, however, are written as they usually are in FORTRAN, even when they apply to the newly-defined data types.

The control structures provided permit the programmer to structure his program in a logical and convenient way, resorting only rarely to the objectionable GOTO statement. Stream input permits him to structure the program physically so that it is most readable.

ACKNOWLEDGMENTS

I would like to thank my supervisor, Robert Zarnke, for suggesting the topic of this thesis, and also for the advice and guidance he provided throughout its development. I would also like to thank Bohdan Zajac for provision of the I/O interface routines used by the processor. His willingness to change them where required was of great help. Finally, I would like to thank Eileen Robbins who spent many hours typing this thesis.

TABLE OF CONTENTS

Chapter 1

INTRODUCTION

The programming language FLEX is an extension of FORTRAN [2] which allows the definition of new data types and new operators, as well as the extension of present operators to include the new data types. It incorporates the four most useful control structures [5] - DO and WHILE loops, CASE and IF selections - which are similar to the corresponding constructs of PL/1 and ALGOL. FLEX also makes other improvements to FORTRAN, such as stream input (as in PL/1) and embedded format specifications in input/output statements.

The ability to define new data types permits the user to define ones which are appropriate for his application, rather than simply using existing data types. He can define new operations on precisely those data types he requires. Thus he is much more specific in the declaration and use of variables than in either FORTRAN or PL/1.

Expressions, however, are written as they usually are in FORTRAN, even when they apply to the newly-defined data types.

The control structures provided permit the programmer to structure his program in a logical and convenient way, resorting only rarely to the objectionable GOTO statement. Stream input permits him to structure the program physically so that it is most readable.

The original motivation for the development of FLEX was the provision of an easy and straightforward way of programming with multi-precise integers. A package for performing operations on multi-precise integers [6] is available in FORTRAN. Operations on such integers are carried out by calling the appropriate functions and function calls may be nested when expressions involving several operations are required.

This implementation of multiple precision arithmetic was not completely satisfactory. Coding even a fairly simple expression resulted in nesting of function calls to many levels. Multi-precise computations were therefore very unreadable and sometimes difficult to debug. Since there were at least two routines corresponding to each operation (one for two multi-precise integers and one for a multi-precise integer and an ordinary integer), it was easy for the user to call a function with arguments of the wrong data types. Both these problems would be solved if operations on multi-precise integers could be written in the usual infix notation used with the ordinary data types. The familiar notation would make programs more readable, and the decision of which routine to call would be made by the processor rather than the programmer.

A preprocessor was needed which would translate operations on multi-precise integers to the appropriate function and subroutine calls. This would then allow the user to code in the notation with which he was familiar. Such a preprocessor would permit multi-precise integers as a new data type and the existing operators would be extended to include the new data type.

The extension of the PL/1 language was considered, one reason being that it contains several desirable control structures. However, the difficulty involved in extending PL/1 to include the new data type and operations was soon apparent. In PL/1, the type of a data item is implied by several attributes and this scheme has no obvious extension for new data types. In addition, the complicated precision rules precluded the adoption of similar rules for multiple precision arithmetic. Finally, if Pl/1 were used, it would have been necessary to modify the multiple precision routines to interface with PL/1.

The extension of FORTRAN was adopted instead. FORTRAN is a relatively simple language and thus easy to produce as the translated (object) code. The data types are determined by a unique mode name (and an optional length field) and are therefore simpler than in PL/1. The "precision rules" are very straightforward; for example, operations on INTEGERs yield INTEGER results. Another benefit was that the multiple precision routines would not have to be changed. FORTRAN, however, does lack many useful features, the most notable being the control structures that are found in PL/1 and ALGOL.

Rather than a special purpose preprocessor to provide only multiple precision operations and variables, a general purpose preprocessor was developed which would handle arbitrary new data types and operations. Thus, multi-precise integers could be included as a special case of this more general mechanism. Other desirable extensions and improvements to FORTRAN have also been incorporated as standard features in the new language. The FLEX preprocessor was designed to translate FLEX programs into FORTRAN, the FORTRAN code subsequently being compiled by a FORTRAN compiler.

FLEX has several new declarations that permit the user to introduce new operators and data types. It also extends the existing statements to permit the declaration of variables of these new types. It contains several other extensions and improvements such as reasonably powerful control structures and input and output statements which permit expressions (of the new as well as the existing data types).

Chapter 2 describes the basic extensions to FORTRAN which have been incorporated in FLEX. Chapter 3 contains a detailed description of all FLEX statements and their syntax. Chapter 4 discusses conclusions reached after having implemented FLEX; it also describes other constructs that might be added naturally to FLEX. Appendix 1 is the complete syntax for FLEX written in BNF. Appendix 2 describes all the predefined information required for the standard FORTRAN data types and operators. Appendix 3 illustrates the running of a sample program with JCL and also contains a list of all the FLEX messages. Appendix 4 contains two programs written in FLEX, together with the generated object code for each.

FLEX was implemented in assembler language [3] on the IBM System /370 computer at the University of Manitoba. Since FLEX translates from one source language into another, it was essential that the language in which it was implemented permit easy manipulation of variable length strings. FORTRAN was rejected immediately on these grounds, as well as software development languages like project SUE system language [8] and PL/360 [9]. It was also desirable that the implementation language permit programmer controlled storage allocation and structured data items so that tables of dynamic size (like the symbol table) might be built and altered easily. Only PL/1 seemed to offer these facilities, but the storage type that seemed most suitable (namely BASED) did not permit VARYING length strings; to have used fixed length strings would have required that the strings be manipulated explicitly by the programmer, as in assembler language. An assembler language program would be much more efficient, both in storage requirements and execution time, than its PL/1 analogue. Thus, assembler language was chosen to implement FLEX although it is extremely machine-dependent as a result.

Chapter 2

LANGUAGE DESCRIPTION

2.1 Introduction

FLEX (FORTRAN Language EXtension)is a preprocessor which translates statements of the source language into FORTRAN. In the following discussion, the acronym FLEX is used to refer to both the source language and the preprocessor.

Originally the FORTRAN language was the model upon which the FLEX syntax was based. FLEX was to be an extension of FORTRAN, incorporating two major improvements, but looking basically like FORTRAN. However, so many changes, improvements and additions were made that most FLEX statements now bear very little resemblance to the original FORTRAN statements.

2.2 Program Structuring

Although structured programs can be written even in a language like FORTRAN, coding is very inconvenient if the appropriate control structures [5] are not available in the language. Thus, one principal aim of FLEX was the introduction into FORTRAN of several reasonably powerful control structures which would simplify the writing of structured programs.

The control structures provided by FLEX are procedures, selections and loops. Each control structure is introduced by a specific statement and ends with a specific terminal symbol. It may contain other clauses that determine control flow.

These constructs make structured programming in FLEX more straightforward than it is in FORTRAN, and, as an added benefit, the FLEX code is more readable than the equivalent FORTRAN would be.

2.2.1 Procedures

As in FORTRAN, there are three different types of routines that the user can code: mainline, subroutine subprogram, and function subprogram. In FLEX these routines are introduced by the PROGRAM, SUBROUTINE and FUNCTION statements respectively and are terminated by the symbol ENDPROC, these three types of routines will be called procedures. The SUBROUTINE and FUNCTION statements also define the procedure name and parameter names, as in FORTRAN.

Declarations made between the beginning statement and the end of a procedure are said to be local to that definition; that is, the declarations are not known in any other procedure. In FORTRAN, all declarations are local to some procedure.

Quite frequently, the user desires to make definitions that apply to several procedures. Such definitions are made before the first PROGRAM, SUBROUTINE or FUNCTION statement appears and are called global definitions. The only statements which may be global are the new declarative statements: MODE, OPERATOR and PRIORITY.

2.2.2 Selections

A selection is a control structure in which a single clause is selected for execution from among several clauses which have been supplied. The clause to be selected is determined by a value which is calculated at execution time. Selections are introduced either by IF or by CASE, and end with ENDIF or ENDCASE respectively.

IF permits two clauses, THEN and ELSE, each of which contains a series of other statements. Selection is based on a logical value.

CASE permits the selection of one of a number of alternative clauses, based on an integer value. As in IF, each clause consists of a series of statements.

2.2.3 Loops

A loop is a control structure in which a series of statements is executed repeatedly, until some condition is satisfied. Loops are introduced either by the WHILE statement or by the DO statement, and end with END.

WHILE permits repeated execution of the series of statements contained in the control structure, until the condition specified in the WHILE statement becomes false.

DO is similar to the FORTRAN DO in that execution is controlled by an index value. Unlike FORTRAN, however, this index is not restricted to integer values. This index assumes values which start with an initial value and proceed to a final (test) value, using a (positive) step value. One or both of the initial and final values may be non-positive. If the user wishes, the initial value may be larger than the final value , in which case, the step value is used as a decrement. Otherwise it serves as an increment. The DO may also be controlled by a WHILE clause. Escape from the loop occurs either when the final value is passed or when the WHILE condition becomes false, whichever occurs first.

2.3 New Modes and Operators

The second major extension of FORTRAN is introduction of the facility to define new data types (modes) and new operators, as well as the ability to extend the domain of present operators to include the new modes.

2.3.1 Mode Relationships

When a new mode is defined in FLEX, a relationship between the new mode and an existing (or "old") mode is established so that translation of FLEX declarations into FORTRAN can occur. A new mode is defined in the MODE statement, in terms of a previously defined mode, and this linking of the two modes is used to determine how the new mode will ultimately be translated. A new mode may be directly related to only one other mode. However, the mode to which a new mode is related can in turn be related to a third mode, and so on. In this way, a mode can be indirectly related to several others.

2.3.2 Fundamental Modes

There are several modes, called fundamental modes, which are not defined in terms of any other mode. These are the FORTRAN modes, INTEGER*2, INTEGER*4, REAL*4, REAL*8, COMPLEX*8, COMPLEX*16, LOGICAL*1 and LOGICAL*4. Every mode must be related, either directly or indirectly, to one of these fundamental modes.

2.3.3 Two Relations

The two types of relations that are possible between a new mode and an existing mode are translation and equivalence.

The translation relation is used to indicate the mode into which a new mode will be translated. The mode into which a new mode is translated may itself be translated, and so on. Variables of the new mode and variables of the translation mode bear no relationship to each other during processing, although they will all be of the same mode in the translated (FORTRAN) code.

When a new mode is declared equivalent to an old mode, the new mode is treated simply as a new name for the old mode. Variables of the new mode are processed exactly as if they were of the old mode. Equivalence is clearly a transitive relation. For example, if mode APPLE is equivalent to mode BEET and mode BEET is equivalent to mode CARROT, then APPLE is also equivalent to CARROT and variables of all three modes are treated as if they were of mode CARROT.

2.3.4 Translation of Modes into FORTRAN

As stated above, every new mode must be related, either directly or indirectly, to a fundamental mode. In the FORTRAN code, the new mode is translated into the fundamental mode to which it is related. Equivalent modes are treated identically during processing; translation modes are treated differently. Suppose, for example, that

a) mode INTB is equivalent to mode INTA,

b) INTA is translated into mode INTEGER, and

c) INTEGER is equivalent to mode INTEGER*4 (as in FORTRAN).

Then, during processing, variables of mode INTB and variables of mode INTA are treated as if they were all INTA variables (because INTB is equivalent to INTA). Similarly, INTEGER and INTEGER*4 variables are all treated as INTEGER*4 variables. On the other hand, INTA variables and INTEGER*4 variables are not of the same mode during processing. However, variables of all four modes will be declared to be INTEGER*4 variables in the resulting FORTRAN code.

2.3.5 Operator Definition

An operator is a function that accepts values (called operands) of specific modes (called the domain) and produces a value (called the result) of some, possibly different, mode. In order to take advantage of new modes, the user must be able to define operators whose domains include these new modes. This process is called "extending" (the domain of) an operator. Extension is unnecessary when the new mode is equivalent to a mode which is already in the domain of the operator ; the existing definitions will be used automatically.

Operators are designated by either:

a) predefined special characters or special character pairs (see 3.1.4), or

b) identifiers.

The user cannot define operators of the first type except for those which already exist. Any new operators must be identifiers.

The extension of an operator is done in the OPERATOR statement. In it, the user defines the modes of the operands (that is, the domain) for which this definition is to be used, these modes being specified in a list, thus establishing an ordering of the operands. In addition, the user defines the mode of the result and a "template" which determines the translation into FORTRAN code (see 2.3.8).

Every operator must have a priority associated with it, so that operations in expressions can be compiled in the proper order. The priority of an operator is specified in the PRIORITY statement. An existing operator already has a priority associated with it, so that only new operators require specification of a priority.

Certain operators have predefined domains and priorities, so that the user can carry out the usual FORTRAN operations on the FORTRAN data types without having to make explicit definitions. A newly-introduced operator has no domain at all, and thus the user must make all relevant definitions.

When the OPERATOR statement extends the domain of an operator, the information contained in it is added to that which has already been defined. For example, if operator PLUS has already been defined to operate on two operands of mode APPLE and is then extended to operate on two operands of mode BEET, both definitions are retained. By successive extensions, any operator can be defined to operate on many different modes, and each individual definition has a unique template associated with it.

2.3.6 Operator Invocation

When an operator is used in an expression, we say it is being invoked. Operators usually have one or two operands which are written in an expression as:

operator operand

operand operator operand .

In FLEX, operators can be defined to have more than two operands. The standard infix form of operator invocation is no longer adequate for this situation. In FLEX, an operator (possibly with more than two operands) can be invoked by specifying a list of operands instead of a single operand. The generalization of infix notation which has been adopted leads to the syntax:

(operand, ...) operator (operand, ...) .

The order of the operands must agree with the ordering established in the definition of the operator. If either operand list contains only one element, then the parentheses can be omitted. When the left operand list is omitted entirely (including the parentheses), the result is prefix notation. The right operand list may never be empty. When each list consists of a single operand, the resulting invocation is in the usual infix notation.

With the exception, noted above, that at least one operand must appear in the right operand list, the operands may appear on either side of the operator, as long as their order is not changed. For example, a ternary operator, say ! , may be invoked in any one of the following ways:

```
!(A,B,C)

A!(B,C)

(A,B)!C
```

where A, B and C are variables of the appropriate modes (as specified in the definition). Notice that the order of the operands remains unchanged in all cases.

2.3.7 Selection of Operator Definitions

An expression is, as usual, a combination of operators and operands. There must be at least one infix operator between every pair of operands. However, each operand can have an arbitrary number of prefix operators, and such prefix operations do not need to be parenthesized.

The order in which the operators are invoked is determined by their priorities. The priorities are just integers which are compared to determine the order of operations. Prefix operators are invoked before infix operators. When a decision must be made about which of two infix operators to invoke, the one with the higher priority is invoked; if they have the same priority then the textually first one is invoked. The result of an operator invocation is (the production of) a new operand which replaces the operator and its operands. This new operand can then be part of a subsequent invocation, at which time, it too is replaced.

Each definition of an operator has a specific number of operand modes specified in a particular order. Any operator invocation has a specific number of operands of specific modes, and the order of the modes is determined by the order of the operands. The processor merely selects the definition of the operator which agrees with the invocation with respect to the number of operands, their modes, and the order in which the modes are specified.

As was discussed previously, each operator definition has a translation template associated with it. When a match is found between the invocation and a specific definition, the corresponding template is used to produce the translation.

2.3.8 Template Expansion

The translation of a particular invocation is produced by substitution of operand values into the template. The template is just a character string which, after substitution, becomes:

a) an optional series of FORTRAN statements, and

b) an optional piece of text (which is a portion of a FORTRAN expression).

The substitution process is referred to as "expansion" of the template. Operator invocation is similar to macro expansion of assembler language.

The statements, if any, are produced as lines of the "object program", that is, the translated version of the user's program. A statement is produced whenever a semi-colon is encountered in the template. Each part of the template which expands into a statement must be followed by a semi-colon. It is the user's duty to make sure that what is produced is a valid FORTRAN statement, because FLEX assumes the template has been correctly specified and does no checking of the code.

The piece of text, if any, which results from the expansion of the template, is passed on and will participate in the expansion of other operators in the expression. In other words, this piece of text is the "result" of the operator invocation. If the operator expands only into FORTRAN statements, the result of the invocation is undefined and thus cannot participate in subsequent invocations.

Items to be substituted into the template are of three different kinds:

a) operand values,

b) a resulting (temporary) variable, and

c) statement numbers.

The quantities in the template, which are replaced by these items, all begin with the escape character "&" . If the user wants an "&" in the translation, he must code "&&" in the template.

The position at which an operand value is to be substituted is identified by "&n" where "n" is a non-zero digit (1-9). "&n" is replaced by the "n"th operand of the current invocation, where the operands are simply numbered from left to right starting at 1. For example, the invocation:

X + Y

has two operands. X is the first operand, and Y the second. If the following definitions have been established:

```
          MODE NEWREAL : REAL ;
          OPERATOR + (NEWREAL,NEWREAL) = NEWREAL
                   ('PLUS (&1,&2)');
          NEWREAL X,Y;
```

then the modes of the two operands in the above invocation match the modes in the operator definition. The result of the invocation would be:

PLUS (X,Y)

which is an operand of mode NEWREAL. This result can then be used in subsequent invocations.

FORTRAN statements cannot be used as operands; that is, it is meaningless to substitute them into another statement or expression. But often the result of an invocation can only be expressed as a series of FORTRAN statements, and not as a FORTRAN expression. To handle such cases, the result can be assigned to "&0", a temporary variable of the resulting (FLEX) mode. The processor creates a temporary variable which is substituted into the template. This temporary can then be used to communicate the result to subsequent invocations. For example, an operator QUERY could be defined as follows:

```
          OPERATOR QUERY (REALA,REALA) = REALA (
                'CALL QUERYS (&0,&1,&2);&0')
```

where REALA is a mode name. Assuming the declaration,

```
          REALA A,B;
```

the invocation:

```
        A QUERY B
```

produces the statement:

```
        CALL QUERYS (T$1,A,B)
```

where T$1 is the name of a temporary. The result of the invocation is:

```
        T$1
```

of mode REALA.

It is sometimes desirable to branch from one statement to another within a particular invocation. FORTRAN statement numbers are required to accomplish this task. To cause substitution of a unique number into the template, the user codes "&Sm" where "m" is a digit (0-9). Thus, up to ten different numbers can be used in a single invocation. These numbers are strictly local to the invocation in which they appear. Invocation of several operators, each of which makes references to "&Sm" , for a particular "m" , would not pose the problem of duplicate numbers because all numbers allocated within a procedure are unique. However, within a particular invocation, each reference to "&Sm" is replaced by the same number. When "&Sm" is the first item in a statement, the number which replaces it is produced in columns one to five of the FORTRAN object statement. Its appearance anywhere else is handled exactly the same as "&n" or "&0" with regard to substitution. The following example will be fully explained in 3.2.9.

```
        MODE VECT:REAL(10);
        OPERATOR + (VECT,VECT) = VECT (
            ' DO &S1 I=1,10;'
            '&S1 &0(I) = &1(I) + &2(I);'
            '&0');
        VECT A,B;
```

The invocation:

```
          A+B
```

might produce the following statements:

```
            DO 1 I=1,10
          1 T$1(I) = A(I) + B(I)
```

and T$1 of mode VECT will be the result. (Note: the actual statement numbers and temporary names for a particular invocation cannot be predicted. The values shown are used as examples only.)

Since "&Sm" is replaced by a unique integer, it may be used in the creation of unique variable names. For example, the above definition could be rewritten:

```
          OPERATOR + (VECT,VECT) = VECT (
              ' DO &S1 I&S2 = 1,10;'
              '&S1 &0(I&S2) = &1(I&S2) + &2(I&S2);'
              '&0');
```

The statements produced by the invocation given above might then be:

```
            DO 1 I2=1,10
          1 T$1(I2) = A(I2) + B(I2)
```

Variable names created in this way are not recognized by FLEX and thus they should be variables of default FORTRAN types.

2.3.9 Definition of Array Modes

The relationship between modes may be a little more complicated than has been indicated so far. The MODE statement can be used to define modes which are arrays. A new mode may be defined as an array of an old mode. In this case, an undimensional variable of the new mode would refer to an array of values of the old mode, and a dimensioned variable of the new mode, to an array of arrays, and so on.

Only the translation relation is valid when array modes are being defined. The new mode is not equivalent to the old mode which appears in the definition; it is equivalent to an array of the old mode. However, the relations hold only between mode names and not arrays. This means that operator definitions should be established for the new mode so that it can be used efficiently, that is, without having to specify the subscripts.

When coding in FLEX, variables of the new mode can be referenced in two ways, with the subscripts which index the array, or without them. If no subscripts are specified, then the variable is treated as an array; that is, its mode is the array mode. If it is subscripted, then the number of subscripts must agree with the number of dimensions used in the MODE definition. The mode of the subscripted variable is the related mode, since the item being selected from the array is of that mode. For example, if the following definitions have been established:

```
MODE VECT : REAL*4(10);
VECT X;
```

then X is a vector of REAL*4 values, and selection of a specific item from the vector means that a REAL*4 value is being selected. In other words, X(I) is REAL*4. However, X is of mode VECT.

The old mode used to define a new array mode may itself be an array mode. Then variables of this new mode inherit the dimensions of each related mode. For example, if the following definitions are added to those already established above:

```
        MODE ARRAY : VECT(10);
        ARRAY Y;
```

then Y is of mode ARRAY, Y(I) is of mode VECT and Y(I,J) is REAL*4. In other words, the number of subscripts specified determines which mode is used during selection of an operator definition.

When several levels of arrays have been defined, as discussed above, the subscripts may be specified in groups to correspond with these levels. As mentioned previously, Y(I,J) is REAL*4 and Y(I) is VECT. To make this relationship clearer, Y(I,J) could be written Y(I)(J). Here again, Y(I) is VECT, and the addition of the second subscript selects a specific REAL*4 value from the total array.

This notation also implies which array items are stored in contiguous storage locations. Just as the items in a REAL*4 vector, say Z, are stored in the order Z(1),Z(2),Z(3),..., the items in W, a vector of mode VECT, are stored in the order, W(1),W(2),W(3),... . But W(I) is itself a vector of REAL*4 values, which are stored in the order, W(I)(1),W(I)(2),W(I)(3),... . In other words, arrays in FLEX are stored in row order. Since arrays in FORTRAN are stored in column order, this means that the dimensions are reversed from those in FLEX. However, this does not affect the coding of the template; subscripts used on variables in the template correspond to the FLEX dimensioning.

2.4 Input and Output

In FLEX, there are four input/output (I/O) statements. The READ and WRITE statements perform formatted (or free format) input and output as they do in FORTRAN. The INPUT and OUTPUT statements are described later in this section. These I/O statements have been extended to include capabilities not available in FORTRAN, as described in the following paragraphs.

a) Formats are included directly with the data item being read or printed, rather than appearing in a separate (FORMAT) statement. In the following statement:

```
WRITE(6)  N:I(5)
```

the data item N will be printed in the first five columns of the line under the format I(5). In the statement:

```
WRITE(6)  :COLUMN(25),  N:I(5)
```

on the other hand, N will be printed starting in column 25 of the print line. The data value is formatted using information supplied in the format item, I(5); its position is controlled by the control item, COLUMN(25).

These two examples illustrate the most common ways in which a format is written. In either case the format or control item is introduced by a colon. The format or control name is usually followed by a list of values appropriate to that item; thus, in I(5), the value 5 is the field width. The format item has the same meaning as the FORTRAN format I5.

Instead of writing just a single format or control item, as illustrated above, a list of items could be written, as follows:

```
WRITE(6)  N:(COLUMN(25),I(5))  .
```

This statement produces exactly the same code as the example given previously. In such a case, of course, only one item in the list may be a format item.

b) In the FLEX I/O statements, the following quantities may be specified as expressions:

i) the unit field, as in:

```
WRITE(I-J)  N:I(5)
```

ii) the data items in an output list, as in:

```
WRITE(6)  N*J-1:I(5)
```

iii) the field widths, etc., in the format and control items, as in:

```
WRITE(6)  :COLUMN(10*(I-J)), N:I(I-J);
```

This is a major extension, since many versions of FORTRAN prohibit the use of expressions for all of the above items.

c) It is possible to READ and WRITE values without specifying a format. This feature is sometimes referred to as "free format".

When a value is written without format specifications as in either of

```
WRITE(6)  N
WRITE(6)  N:COLUMN(10)
```

a default format is used that depends on the mode of the variable being printed. This default will permit all possible values of the given mode to be written.

When a value is read without format specification, the input records are searched from the current position until a non-blank field is found. This non-blank field is then read as the value of the item. Under "free read", values in the input are separated either by blanks or by commas; if two successive commas are used then a value for the corresponding item of the input list is not read. For example, the data card corresponding to the statement:

```
READ(5)  A,B,C
```

might be any of the following:

```
1    2      3

1, 2, 3

1  ,  2 ,   3

1, , 3
```

While it is possible to mix items with and without formats in the same READ statement, the user should exercise great care when doing so because of the radically different way that such input works. To describe how they interact, it is sufficient to indicate where the field of an input item ends:

i) the field of an item with a format ends after the number of characters specified in the field width;

ii) the field of an item without a format ends at (and includes) the blank or comma immediately following it, or at the end of the current record. For example, the READ statement:

```
READ(5)  I,J:I(5),K,L:I(2)
```

will accept as input any of the following:

```
3 _ | _ _ _ _ 8 |      15 _ | _ 0 |

             3 , | _ _ _ _ 8|    15 _ | _ 0 |
```

where the bars indicate the ends of the fields, and the underscores indicate the spacing required by the statement.

d) FLEX DO clauses have been included in the I/O statements to control input and output much as the FORTRAN I/O loops do. A typical I/O loop would be coded in FLEX as follows:

```
READ(5) DO(I=1 TO N, A(I):E(10,6));
```

The syntax of the FLEX I/O loop differs from that of the FORTRAN loop. Instead, the loop obeys the rules of the FLEX DO clauses.

Within each I/O loop the user may have a list of variables to be read or values to be printed, as in:

```
READ(5) N, DO(I=1 TO N, A(I), B(I), C(I));
```

Another I/O loop may be included, as is permitted in FORTRAN. For example:

```
READ(5) N, DO(I=1 TO N, DO(J=1 TO N, A(I,J)));
```

The I/O loops may include any clauses that are used in normal DO clauses. For example:

```
READ(5) N, DO(I=1 TO N WHILE X(I)¬=0.,X(I));
```

e) In FLEX, a new I/O statement does not automatically cause a new card to be read or a new print line to be started, as it does in FORTRAN. By omitting the unit number in an I/O statement, the user indicates that the data items in the statement are to be read or printed continuing from the position in the card or print line where it stopped in the previous I/O statement (which must be of the same type as the current one). The following example shows one application of this feature:

```
READ(5) I:I(2);
IF I=1 THEN READ, J:I(10)
       ELSE READ, AJ:F(10,6) FI;
```

Here the second field (columns 3-12) is either an integer or a real value depending on the integer in columns 1 and 2.

FLEX I/O statements are translated into a series of FORTRAN CALL statements. The subroutines which are called carry out both the formatting and control functions specified in the I/O statement. Each mode defined in a FLEX program may have up to four subroutines associated with it, each subroutine corresponding to one of the I/O statements. For example, suppose the user wishes to define a new mode that is printed by calling a special subroutine, OUTMOD. He writes:

```
MODE NEWMODE = REAL*4(10) WRITE(OUTMOD);
```

In order to read or print a value of a particular mode, a call is made to the appropriate subroutine which is selected from those provided for that mode.

The user need not specify I/O routines for the new modes that he defines. In this case, if values of the new mode are to be read or printed, the routines associated with its related mode are used. Every fundamental mode has a subroutine associated with it for each type of I/O statement. Thus an appropriate routine can be found to read or print values of any mode. A description of the predefined routines is given in Appendix 2.

The INPUT and OUTPUT statements transfer data directly from storage to a file without any conversion whatsoever; they are the analogues of the FORTRAN "unformatted" READ and WRITE statements. They are intended to allow the user to save and retrieve data on auxiliary storage without incurring the expense of conversion to printable character strings. No format items are used in these statements.

This chapter has discussed the main additions and changes that have been made to FORTRAN. The next chapter describes the statements of the language in detail.

Chapter 3

STATEMENT DESCRIPTION

3.1 Metalanguage

3.1.1 Production Rules

A modified version of the metalanguage BNF [1] is used to describe the syntax of the language FLEX. The representation of FLEX syntax uses production rules, which are composed of terminal and non-terminal symbols. A terminal symbol is a sequence of characters which is recognized as a unit during lexical analysis. A non-terminal symbol, on the other hand, is the name of a sequence of terminal and non-terminal symbols.

A production rule is written:

$$a = w$$

where "a" is a non-terminal symbol and "w" is a sequence of terminal and non-terminal symbols, called an expansion of "a" . Whenever "a" has several alternative expansions, the production rules defining "a" are abbreviated as follows:

$$\begin{aligned} a = {} & w_1 \\ & w_2 \\ & \cdot \\ & \cdot \\ & \cdot \\ & w_R \end{aligned}$$

where the w_i are sequences of terminal and non-terminal symbols.

3.1.2 Character Set

The characters of the source stream are divided into a number of classes designated by the symbols, "letter", "digit", "special character", "punctuation" and "blank". These classes are defined as follows:

letter A-Z $ # @ _

digit 0-9

special character ¢ < + ¦ & ! * ¬ / > ? =

punctuation . ( ) ; , % : ' "

blank ƀ

"Special character" is distinguished from "punctuation" because only special characters may form part of an operator.

3.1.3 Non-terminal Symbols

A non-terminal symbol is designated by its name which is a sequence of lower-case letters and hyphens. Several conventions have been adopted in the naming of non-terminal symbols. These conventions are described below.

If "x" is the name of a non-terminal symbol, then the non-terminal symbol, "x-list", is defined by:

```
x-list = x
         x-list, x
```

That is, "x-list" designates a sequence of "x"s separated by commas.

If "x" is the name of a non-terminal symbol, then the non-terminal symbol, "x-sequence", is defined by:

```
x-sequence = x
             x-sequence x
```

That is, "x-sequence" designates a sequence of "x"s with no separating punctuation.

If "x" is the name of a non-terminal symbol, then the non-terminal symbol, "x-series", is defined by:

```
                x-series = x
                           x-series;x
```

That is, "x-series" designates a sequence of "x"s separated by semi-colons.

If "x" is the name of a non-terminal symbol, then the non-terminal symbol "x-option" designates the optional occurrence of the non-terminal "x".

In this chapter, production rules of the forms discussed above have been omitted from the syntactic definition of each statement. However, the full language definition given in Appendix I includes all rules of the above forms.

3.1.4 Terminal Symbols

In a production rule, a terminal symbol is denoted either by the appropriate punctuation or special character, by upper-case letters, or by the name of a class of terminal symbols as described below.

There are several symbols which, although designating classes of terminal symbols, are treated as terminal symbols themselves during lexical analysis. These built-in terminal symbols are defined by the following "production rules":

```
identifier = letter
             identifier letter
             identifier digit

literal = 'charstring'
          "charstring"
```

```
integer = digit
          integer digit

operator = identifier
           specialcharacter
           <= || ** ¬< ¬> ¬= -> // >= :=

empty =

constant = integerconstant
           shortintegerconstant
           realconstant
           longrealconstant
           imaginconstant
           longimaginconstant
           logicalconstant

integerconstant = sign-option integer

shortintegerconstant = integerconstant S

realconstant = decimalconstant realexponent-option
               integerconstant realexponent

longrealconstant = decimalconstant longrealexponent
                   integerconstant longrealexponent

imaginconstant  = integerconstant I
                  realconstant I

longimaginconstant  = longrealconstant I

logicalconstant = TRUE
                  FALSE

decimalconstant = integerconstant . integer-option
                  sign-option . integer

realexponent = E integerconstant

longrealexponent = D integerconstant
```

3.2 Input Conventions

3.2.1 Use of Terminal Symbols

The source program is a sequence of the terminal symbols introduced above. The sole use of the character "blank" is to separate other terminal symbols in the input stream. It does not form a part of any other terminal symbol. Successive terminal symbols may be separated by an arbitrary number of blanks. In the case of adjacent identifiers, adjacent literals or adjacent integers, at least one blank must be inserted to avoid ambiguity. Where two operators are adjacent, blanks may have to be inserted in order to avoid confusion with the double character operators (see 3.1.4).

3.2.2 Card Boundaries

Input records for FLEX are 80 characters long. Columns 1 through 72 contain source text; columns 73 through 80 are ignored and may be used for sequence numbers. In other words, column 73 acts effectively as the card boundary. With a few exceptions, card boundaries have no effect on the input stream. This means that continuation from one card to the next is automatic. The first exception is that a single terminal symbol may not be continued across a card boundary; that is, identifiers, literals, constants and operators must appear entirely on one card. The second is that a "card comment" may not be continued.

3.2.3 Comments

There are two types of comments in FLEX. One is a statement (the COMMENT statement) which is described in 3.7. The second is the card comment mentioned above. This comment begins with the punctuation symbol "%" and ends at the card boundary. A card comment may appear anywhere in a program. If desired, it may start in the middle of a statement. It has no effect on the source program. In fact, the "%" symbol has the same effect as the card boundary. Whenever either is encountered, scanning is continued on the next card. If a "%" is used in column 1 of a card, the entire card is a comment.

3.2.4 Literal

A literal is bounded by two identical quote symbols. If the user wishes to include the bounding quote character in the literal, he duplicates it as in PL/1. Because "%" by itself is used to indicate the card boundary, a "%" is included in a literal by writing "%%" . A literal may not be continued on the next card.

3.2.5 Reserved Words

In FORTRAN, a statement is not recognized merely because it starts with a keyword. The syntax of the statement is also considered. Thus the statement:

```
DO 10 I=1
```

is interpreted as an assignment statement. Similarly:

```
WRITE (I,J) = B
```

is not a WRITE statement. The use of keywords as variable names can be very confusing to the reader of a program, but it also makes the statement recognition process more complicated for the processor. In FLEX, keywords and "punctuation" words are reserved for the specific uses described in the syntax, and they cannot be used elsewhere.

In the description given above, the term keyword is used to refer to those special identifiers that appear at the beginning of a statement and identify its type. Some keywords in FLEX are: PRIORITY, DO, and FUNCTION, and they identify the PRIORITY, DO and FUNCTION statements respectively. If the user has a syntax error in, say, a DO statement, FLEX considers it to be an incorrect DO statement rather than an undecodable statement or assignment statement, as in FORTRAN.

A punctuation word is an identifier which appears in a control structure for the purpose of delimiting specific parts. Examples of punctuation words are: DOWNTO, IN, OUTCASE and ENDPROC.

All punctuation words and keywords are reserved. Reserved words are indicated in the syntactic definition as terminal symbols. However, not all terminal symbols which are words are reserved. The format and control names used in the I/O statements are not reserved.

A few of the keywords and punctuation words have alternate forms (synonyms). A list of these synonyms may be found in Appendix 1.

3.2.6 The Semi-colon

The semi-colon is used as a statement separator rather than a terminator. Semi-colons are used between adjacent statements or control structures. Although a series of control structures and statements (see 3.1.4) does not require a semi-colon at the end, it is not incorrect for one to be inserted. Any statement which is immediately followed by a punctuation word does not have to end with a semi-colon; the punctuation word fulfills its function. As indicated above, however, any control structure which is immediately followed by a new statement must be separated from that statement by a semi-colon.

3.3 Source Program

The general structure of the source (user's) program is shown below:

```
sourcepgm = globaldefn; proc-series
            proc-series

globaldefn = defnstmt-series

defnstmt = modestmt
           priostmt
           operstmt
           cmtstmt
```

```
proc = pgmstruc
       fcnstruc
       subrstruc
```

The program may start with a series of global definition statements, which are separated from the following procedures by a semi-colon. Only new modes, operators and priorities may be defined in this global area. Any definitions which are made in the global definitions apply to every procedure in the program and do not have to be made in each procedure which uses them.

There are three types of procedures: the mainline, the subroutine and the function. As in FORTRAN, only one mainline procedure should be supplied. However, there is no restriction on the number of functions or subroutines that may appear. The procedures may be specified in any order that the user desires but they may not be nested.

3.4 Control Structures

3.4.1 General Structure

A description of the various control structures which have been implemented in FLEX is given below:

```
pgmstruc = pgmstmt; procstmt-series ENDPROC

fcnstruc = fcnstmt; procstmt-series ENDPROC

subrstruc = subrstmt; subrstmt-series ENDPROC

dostruc = dostmt; procstmt-series-option END

whilestruc = whilestmt; procstmt-series END

casestruc = CASE caseclause ENDCASE

ifstruc = IF ifclause ENDIF
```

The general structure (an introductory statement followed by a series of other statements and terminated with a special punctuation word) is quite straightforward. The slightly more complicated cases of the selection structures, IF and CASE, are described later.

The types of statements and structures allowed in the "body" of a control structure are described as follows:

```
procstmt = labeldefn-sequence-option imperative
           declstmt
           defnstmt

subrstmt = procstmt
           entrystmt

labeldefn = label :

label = identifier

imperative = loop
             selection
             stmt

loop = dostruc
       whilestruc

selection = casestruc
            ifstruc

stmt = assignstmt
       iostmt
       callstmt
       stopstmt
       returnstmt
       gotostmt
```

There are two basic constructs which may appear in any control structure:

a) definition and declaration statements, and

b) labelled imperatives.

In addition, the subroutine procedure may also contain the ENTRY statement which is not allowed in any of the other structures.

Definition statements (namely, MODE, PRIORITY and OPERATOR) provide information for the proper processing of the other statements. Such statements generate no object code but merely control the translation process. Definition statements may appear in the global definition area as well as locally in the various procedures.

Declaration statements are statements in which variables are declared. They may be used locally in a procedure, but not globally.

Both of these types of statements, as well as the procedure statements (namely, PROGRAM, FUNCTION, SUBROUTINE and ENTRY), should not be labelled.

The remaining control structures as well as all statements in FLEX which have not yet been mentioned, are referred to as imperatives, all of which can be labelled. The control structures of this type are loops and selections. The three most frequently used statements which appear in this classification are the assignment, input/output and CALL statements.

Because labels are identifiers, meaningful names can be chosen to help document the program. Of course, if the program is structured, GOTO's (which use labels) will be used infrequently.

As indicated previously, procedures cannot be nested. However, all other control structures may be. The ability to nest loops and selections is a very powerful programming tool.

3.4.2 Mainline Procedure

The PROGRAM statement introduces the mainline procedure. Its syntax is as follows:

```
pgmstmt = PROGRAM
```

The procedure ends with the punctuation word ENDPROC. Execution of a FLEX source program begins when the system transfers control to (calls) the mainline procedure (as in FORTRAN) and ends when control reaches the ENDPROC of the mainline procedure (or a STOP statement).

3.4.3 Function Procedure

A function procedure is introduced by the FUNCTION statement and ends with ENDPROC. This introductory statement has the following syntax:

```
fcnstmt = FUNCTION resultmode fcname (parm-list)
resultmode = modename
fcname = identifier
parm = identifier
```

The result of a function is returned to the calling procedure in the same way as in FORTRAN: the value is assigned to the function name. The mode of the result is defined in the FUNCTION statement, as are the parameter names.

The FLEX FUNCTION statement is reproduced essentially unchanged in the object program; that is, a FORTRAN FUNCTION statement is produced whose function name and parameters are taken directly from the FLEX statement (although they may be truncated to conform to FORTRAN naming conventions), and whose mode is the fundamental mode that is related to "resultmode".

Parameters are declared in declaration statements within the function, and these declarations are translated exactly as are variable declarations.

The resulting mode and the modes of the parameters are known only within the function procedure itself, that is, these modes are not used for translating function calls, which occur only in other procedures (see 3.6.1).

In order to properly translate function calls, the user must make a global definition (in an OPERATOR statement) which specifies the modes of the parameters and the mode of the result of the function (see 3.5.4). The correctness of a function call, with respect to the number and modes of the arguments, is determined by matching the modes of the arguments with the modes in this definition in a process exactly like that used in operator invocation.

The resulting mode may not be related to an array mode, because of restrictions on the types of values that may be returned from a FORTRAN function. Thus the following sequence of statements would be incorrect:

```
MODE REALA : REAL(25);
OPERATOR QUERY (REALA,REALA) = REALA;
FUNCTION REALA QUERY (A,B);
REALA A,B;
```

However, the desired result, of invoking a function which returns an array of values, could be achieved using the following sequence of statements:

```
MODE REALA : REAL(25);
OPERATOR QUERY (REALA,REALA) = REALA
     ('CALL QUERY (&0,&1,&2); &0');
SUBROUTINE QUERY (C,A,B);
REALA A,B,C;
```

An acceptable invocation of the operator/function would be:

```
QUERY (A,B)
```

The difference between invoking an operator which produces a subroutine call, and calling a subroutine procedure, is that the operator invocation may occur in an expression while a subroutine call is a separate statement.

3.4.4 Subroutine Procedure

A subroutine procedure is introduced by the SUBROUTINE statement and ends with ENDPROC. The syntax of the SUBROUTINE statement is:

```
subrstmt = SUBROUTINE procname (parm-list)
procname = identifier
parm = identifier
```

A subroutine procedure is called in a separate statement, the CALL statement (see 3.6.4). Values calculated in a subroutine may be returned via the parameter/argument lists as in FORTRAN. Thus any values which are returned to the calling procedure do not enter into further computations immediately, as does the result returned by a function procedure.

As is the case for functions, the modes of the parameters are known only in the subroutine procedure. In order to determine the correctness of subroutine calls with regard to both the number of parameters and their modes, the information must be known to the other procedures (which call the subroutine). It is declared in a (global) OPERATOR statement (see 3.5.4), so that a matching process can be carried out as it is for operator invocation.

The FLEX SUBROUTINE statement is translated in the object program exactly as it is coded in the source program, with the possible exception that truncation of the procedure or parameter names could occur so that they conform to FORTRAN naming conventions.

3.4.5 ENTRY Statement

The ENTRY statement may occur only in a subroutine procedure. It may not appear within any other control structure. Its syntax is:

```
entrystmt = ENTRY procname (parm-list)
```

It establishes a secondary "entry point" to the procedure. An entry point is the place in the procedure to which control is transferred when the procedure is called. Thus a call using an entry name rather than the subroutine name would transfer control to the appropriate entry point within the procedure rather than to the beginning of the procedure.

The parameter list in an ENTRY statement need not have the same names or number of parameters as parameter lists in the SUBROUTINE statement or in other ENTRY statements occurring in the same procedure. A parameter must be declared and used <u>after</u> the first ENTRY statement in which it appears. Thus the following sequence of statements uses parameters P3 and Q incorrectly:

```
SUBROUTINE ABC (P1,P2);
REAL P1,Q;
INTEGER P2;
P1 := 2**P2-P3;
ENTRY E1 (P3,Q);
```

With the possible exception that truncation of the entry name or parameter names might occur so that they conform to FORTRAN naming conventions, the ENTRY statement is translated in the object program exactly as it is coded in the source program.

To ensure that calls to a secondary entry point are translated properly, the parameter information must be declared in a (global) OPERATOR statement, just as the primary entry point (the beginning of the procedure) is declared.

3.4.6 If Selection

An IF control structure is introduced by the keyword IF and ends with the punctuation word ENDIF. It has the following syntax:

```
ifstruc = IF ifclause ENDIF

ifclause = expr thenclause elseclause-option

thenclause = THEN procstmt-series

elseclause = ELSE procstmt-series

             ELSEIF ifclause
```

In an IF selection, one or two clauses are provided, one of which is selected for execution based upon the value of the expression in the selection. If the value is true, the THEN clause is selected; otherwise, the ELSE clause is selected. If an ELSE clause is not provided and the value is false, then an immediate exit from the selection is taken.

If the else clause consists solely of another IF selection, the user may introduce the second IF by ELSEIF. When this is done, the second IF selection is automatically terminated by the ENDIF of the first IF. If many IF selections are coded in this way, all will be terminated by the ENDIF which corresponds to the original IF, as shown in the following example:

```
IF     I=1 THEN
     A:=B;
     I:=3
ELSEIF  I=3 THEN
     A:=C-B;
     I:=10
ELSEIF  I=10 THEN
     A:=C/B;
     I:=1
ENDIF
```

This type of program structuring is quite common, and the ELSEIF feature helps reveal the structure. In addition, superfluous ENDIFs are avoided, making the program more readable.

The IF selection may be extended by the user as described below. An IF selection like:

```
IF I=0 THEN A:=0
       ELSE B:=0 ENDIF
```

is translated into code similar to the following:

```
  IF(.NOT.(I.EQ.0))GO TO 10
  A=0
  GO TO 11
10 B=0
11 ...
```

Here it is clear that the operator ¬ has been applied to the original expression (I=0) to obtain the result used in the object code. Since the ¬ operator has the following definition:

```
OPERATOR ¬(LOGICAL) = LOGICAL('.NOT.&1')
```

the above code is produced. However, if the user wishes to define the ¬ operator appropriately, he may use any type of expression in the IF. This is illustrated in the following example:

```
OPERATOR ¬(INTEGER) = LOGICAL('&1.EQ.0');
INTEGER I;
I:=2;
IF I THEN A:=B
     ELSE A:=C ENDIF
```

In this case, the THEN clause will be selected. In fact, it will be selected whenever I has a non-zero value. In order that the object code be correct, the mode of the result of a ¬ invocation must be related to a LOGICAL mode. However, as seen above, the mode of the original expression can be anything at all, as long as the appropriate operator extensions have been made.

3.4.7 Case Selection

A CASE control structure is introduced by the keyword CASE and ends with the word ENDCASE. It has the following syntax:

```
casestruc = CASE caseclause ENDCASE
caseclause = expr inclause orclause-sequence-option outclause-option
inclause = IN procstmt-series
orclause = OR procstmt-series
outclause = OUT procstmt-series
            OUTCASE caseclause
```

The following simple example illustrates the general structure of a CASE selection:

```
CASE I IN
     A:=B;
     I:=2
OR   A:=C-B;
     I:=3
OR   A:=C/B
     I:=4
OUT  A:=B*C
ENDCASE
```

In a CASE selection, several clauses are provided, at most one of which is selected for execution. Each clause consists of a series of (labelled) imperatives, declaration statements and definition statements as described in 3.4.1. A clause is selected using the integer value of the expression in the selection. If the value is 1, then the first clause is selected; if it is 2, the second clause is selected; and so on. The clauses are separated by the word OR, and the entire sequence of clauses is introduced by the word IN. Any number of these clauses may be specified, as long as there is at least one. The user can supply another clause, the OUT clause, which will be selected when the integer value lies outside the range for which specific clauses have been provided. If no OUT clause is provided and the value is outside the range, then none of the clauses in the selection will be executed. For example, if five clauses have been provided, then any value not in the range 1 - 5 will cause selection of the OUT clause (or exit from the selection).

If the OUT clause consists entirely of another CASE selection, it may be introduced by the keyword OUTCASE. If this is done, the second CASE selection does not require a terminating ENDCASE; it will be terminated automatically by the ENDCASE corresponding to the first CASE. Many CASE selections could be nested using this method, and only one ENDCASE would be required to terminate them all. If the user wishes to write the second CASE selection separately, he may do so but then the second CASE requires an explicit ENDCASE of its own. This is similar to the nesting of IF's using ELSEIF.

The CASE selection may be extended by the user in much the same way as the IF selection can. A CASE selection such as:

```
          CASE I  IN
               A:=B
          OR   A:=B*C
          OUT  A:=B/C
          ENDCASE
```

would be translated into code similar to:

```
             ITEMP1 = I
             GO TO 10
          12 A=B
             GO TO 11
          13 A=B*C
             GO TO 11
          10 GO TO (12,13), ITEMP1
             A=B/C
          11 ...
```

The expression which appears in the selection is assigned to an INTEGER temporary. The assignment operator (:=) has been predefined to allow all the FORTRAN mixed-mode assignments. If the user wishes to use an expression which normally cannot be assigned to an INTEGER, he must make the appropriate extension of the := operator. For example, the expression could be logical, as illustrated below:

```
          OPERATOR :=(INTEGER,LOGICAL)=
                 ('&1=0;'
                  'IF(&2)&1=1;');
          LOGICAL L;
          L := TRUE;
          CASE L IN
               A:=B
          OUT  A:=B/C  ENDCASE
```

In this example, the first clause will be selected.

3.4.8 While Loop

A while control structure is a loop which is controlled by a logical value. The while loop is introduced by the WHILE statement and ends with the word END. The WHILE statement has the following syntax:

```
whilestmt = WHILE expr
```

In general, the expression in the statement should translate into a logical expression. However, as in the IF selection, if the ¬ operator is defined appropriately, the expression may be of any mode.

The following is a typical while loop in FLEX:

```
WHILE ABS (SUM2-SUM1) > EPS;
   H:=H/2;
   SUM1:=SUM2;
   SUM2:=(APB + 4*S4 + 2*S2)*H
END
```

The loop is executed as long as the expression remains true. If, when it is tested, the value is false, execution of the loop is terminated. The expression is tested at the beginning of the loop; thus, if it is false to begin with, the loop will not be executed at all.

3.4.9 Do Loop

A do control structure is a loop which is controlled by an index which is updated after each iteration. The do loop is introduced by the DO statement. This statement provides the information necessary to control the loop. It has the following syntax:

```
dostmt = DO initclause controlclause

initclause = varname:=expr

controlclause = testclause byclause-cption

                testclause-option byclause-option whileclause

testclause = TO expr

             DOWNTO expr

byclause = BY expr

whileclause = WHILE expr

varname = identifier
```

The loop is terminated with the word END. The following example illustrates a typical do loop:

```
DO X:= -5 TO 5 BY .5;

   WRITE , X, F(X)

END
```

The variable name which appears in the initialization clause is generally referred to as the "index". As in FORTRAN, the index may not be subscripted. However, the value with which it is initialized may be any expression, rather than just a simple constant or variable. The value of the initial expression may be positive, negative or zero.

The test clause may be specified in two ways, with the word TO or the word DOWNTO. When TO is specified, the step value is used as an increment, and the loop ends when the value of the index is larger than the test value. In the following DO statement:

```
INTEGER I;

DO I:=1 TO 100;
```

the index I takes on all values from 1 to 100 in steps of 1. When it attains a value of 101, control passes from the loop. On the other hand, when DOWNTO is specified, the step value is used as a decrement and the loop ends when the value of the index is smaller than the test value. In the statement:

```
        DO I:=100 DOWNTO 1 BY 6;
```

I starts with the value 100 and takes on every sixth value down to 4. When it reaches the value -2, which is beyond the range for which the loop is defined, execution of the loop is terminated.

The index is updated each time the loop is executed. The "step size" used to update the index is supplied in the BY clause; if no BY clause is given, the integer value '1' is used. Escape from the loop occurs when the value of the index is beyond the value which is supplied in the test clause. Comparison of the index and test values occurs at the beginning of the loop. Thus the statements in the loop controlled by:

```
        DO I:=5 TO 4;
```

are not executed at all, since the index starts out with a value greater than the test value. If no test clause is specified, a WHILE clause must be supplied in order to provide an exit from the loop.

The values which are given in the test and BY clauses may be expressions. The test value may be positive, negative or zero. However, the step value should always be positive. If it has a zero or a negative value, an infinite loop may result. For example, the statement:

```
        DO I:=5 TO 10 BY -1;
```

will result in an infinite loop, since the index can never attain the test value by repeated increments of -1. Thus, using a negative step value with a TO clause is not equivalent to using a positive step value with a DOWNTO clause, because the type of comparison that is made is different in the two cases. For example,

```
DO I:=5 TO 1 BY -1;
```

is not equivalent to:

```
DO I:=5 DOWNTO 1;
```

In the first case, the loop will never be executed because the initial value is greater than the test value. The loop controlled by the second statement will be executed five times.

The WHILE clause provides an alternate exit from the loop. While the expression specified in the clause remains true (and the index has not gone beyond the test value), execution of the loop will continue. If the expression becomes false, the loop ends. For a discussion of the mode of the expression in the WHILE clause, consult the description of the while loop (3.4.8) and the IF selection (3.4.6).

The following do loop illustrates the use of the WHILE clause:

```
DO I:=1 TO 10000 WHILE X1-X2 < EPS;
   X1:=X2;
   N:=N*2;
   X2:=F(X,N)
END
```

Although the WHILE clause generally accompanies a test clause,it can appear alone. However, such cases should generally be replaced by the WHILE statement.

The do loop given above would normally be translated as follows:

```
      I = 1
      T1 = 10000
      GO TO 10
   11 I = I+1
   10 IF(I.GT.T1) GO TO 12
      IF(.NOT.(X1-X2.LT.EPS)) GO TO 12
      X1 = X2
      N = N*2
      X2 = F(X,N)
      GO TO 11
   12 ...
```

where the temporary T1 is of the same mode as the index I . If an explicit step value had been specified, it too would have been assigned to a temporary variable of the same mode as I . Of course, if a DOWNTO clause had been used instead of the TO clause, the index would be decremented (I=I-1 in line 11) by the step size and tested (I.LT.T1 in line 10) against the test value. In either case, when the index is updated, the assignment operator (:=) is invoked. In order to extend the do loop so that new modes can be used, the user must extend the appropriate operators. If a test clause is not supplied, the relational operators (> and <) need not be extended. In this case, the step value is always used as an increment, so only the + and assignment operators need actually be extended.

The expression in the WHILE clause can be of any mode, as described in previous sections. If the := operator is extended appropriately, the other expressions in the DO statement can also be of any desired mode. In general, the standard FORTRAN modes, INTEGER and REAL, are used because all of the regular operators have been predefined for these modes.

3.5 Definition and Declaration Statements

3.5.1 Mode Names

In FLEX, as in FORTRAN, the name of a mode comprises two parts:

```
modename = identifier submode-option
submode = * identifier
          * integer
```

Unlike FORTRAN, however, the submode has no intrinsic meaning (such as indicating the amount of storage occupied by variables of that mode), but is used merely as a second level of identification, thus allowing many different modes to be grouped under one general name. For example, the user may define three <u>different</u> mode names, INTEGER, INTEGER*2 and INTEGER*M. This illustrates that INTEGER, without an explicit submode, is a mode which is different from INTEGER*2 which does have an explicit submode (the user should consider INTEGER as having a submode which is different from all other explicitly-specified submodes).

3.5.2 MODE Statement

The MODE statement is used to define new mode names or mode arrays, and also to give the subroutine names which should be used for the input and output of values of the new modes. It has the following syntax:

```
modestmt = MODE modelt-list
modelt = modedefn ioelt-sequence-option
modedefn = modename : oldmode
           modename = oldmode
oldmode = modename dimelt-option
dimelt = (dim-list)
dim = integer
```

```
ioelt = ioname (iortn)
ioname = INPUT
         OUTPUT
         READ
         WRITE
iortn = identifier
```

A new mode is defined by relating it to an old mode, or to an array of an old mode. If the new mode is to be treated as a different mode than the old mode during processing, then the translation relation is used:

```
MODE INTEGER*M : INTEGER     .
```

If the new mode is to be just a new name for the old mode, the equivalence relation is used:

```
MODE NEWREAL = REAL     .
```

No matter which of these two relations is actually specified when defining an array mode, translation is always assumed:

```
MODE VECT : REAL(25)     .
```

As noted in the syntax description given above, more than one mode definition can be made in a MODE statement:

```
MODE INTEGER*M : INTEGER, NEWREAL = REAL     .
```

The user should read Chapter 2 for a more detailed presentation of the concepts involved in this statement.

The "ioelt" permits the user to declare the names of routines which are to be called in order to perform input or output operations on values of the given mode. Up to four subroutine names can be specified, each one corresponding to one of the four I/O statements. If no special subroutines are to be used for input and output of the values of the new mode, the routines corresponding to its related mode are used. All the FORTRAN modes have subroutine names associated with them for all four I/O operations, and these will be used if the user does not wish to define his own I/O routines.

To define a new mode NEWMODE as an array of twelve REALs, in which READ and WRITE statements are to be supported with special subroutines IONMR and IONMW, respectively, the user would code the following statement:

```
MODE NEWMODE : REAL(12) READ(IONMR) WRITE(IONMW)   .
```

3.5.3 PRIORITY Statement

The function of the PRIORITY statement is to assign a priority to an operator, and thus to rank it relative to the other operators. As indicated in the syntax description below, there are two ways in which the value of a priority can be established:

```
priostmt = PRIORITY prioelt-list
prioelt = operator = prio
prio = integer
       operator
```

The first method provides an actual value:

```
PRIORITY QUERY = 150;
```

In order to use this method, the user must know the actual priorities assigned to the old operators in order to choose a value which correctly ranks the new operator with respect to the old operators.

The second method can only be used if the new operator is to have the same priority as some existing operator:

```
PRIORITY QUERY = + ;
```

In this case, the (possibly unknown) integer value which indicates the relative priority of + is assigned to the new operator QUERY.

The integer value of the priority permits the operator to be ranked relative to all other operators. During the compilation of an expression, an operator with a high priority will be invoked before one of lower priority, when they both could apply to the same operand. Thus, in:

```
PRIORITY +=600, *=700;
A:=B+C*D
```

the operand C could either be added to B first or be multiplied by D. Since * has higher priority, the multiplication is done first. If the two operators have the same priority, the one on the left is done first. Thus, if the statements:

```
PRIORITY QUERY = + ;
A:=B QUERY C*D+E
```

are added to the above, the expression is treated as if it had been parenthesized as:

```
A:= (B QUERY (C*D)) + E    .
```

Thus, the actual value of a priority is of little importance, what is important is its relation to the other priorities.

The priority scheme which exists for the predefined operators is given in Appendix 2. Included with the relative ordering of the operators are the actual priority values used. The predefined values were used in the definition of + and * given above.

3.5.4 OPERATOR Statement

The OPERATOR statement has several functions:

a) to define new operators,

b) to extend the domains of existing operators,

c) to declare FLEX functions, and

d) to declare FLEX subroutines.

It is a very important statement because the definitions established by it are used during the processing of the entire FLEX program. If these definitions are established incorrectly, the entire program will be affected. Thus, the user should exercise care when coding an OPERATOR statement.

The OPERATOR statement has the following syntax:

```
operstmt = OPERATOR operelt-list
operelt = operdecl
          fcndecl
          subrdecl
operdecl = operator (modename-list) = resultmode-option template
fcndecl = fcname (modename-list) = resultmode
subrdecl = procname (modename-list)
resultmode = modename
template = (literal-sequence)
```

3.5.4.1 Operator Declaration

The extension of existing operators or the definition of new ones is accomplished in an "operator declaration". Such a declaration provides the information required by FLEX to translate an invocation of the operator correctly. A specific operator may appear in many different declarations. Since the use of an operator with operands of specific modes is to be translated uniquely, the list of modenames should be different in each declaration. There could be a different number of modes, they could be specified in a different order, or the modes themselves could be different.

For example, the operator ** could be defined as follows:

```
          OPERATOR ** (INTEGER) = REAL('EXP(FLOAT(&1))'),
                   ** (REAL) = REAL('EXP(&1)'),
                   ** (INTEGER,INTEGER) = INTEGER('&1**&2'),
                   ** (REAL,INTEGER) = REAL('&1**&2'),
                   ** (REAL,REAL) = REAL('EXP(ALOG(&1)*&2)');
```

the last three lines being standard definitions.

The list of modenames supplies several pieces of information for a particular declaration:

a) the number of operands required by the operator,

b) the modes of these operands, and

c) the order in which they must appear.

This information is used in selecting the appropriate template (as described in Chapter 2). Referring to the example above, if X is a REAL variable, then:

```
          Y:=**X
```

would produce the following translation:

```
          'Y = EXP(X)'
```

while a binary invocation like X**J, where J is INTEGER, would produce a REAL result of the form:

```
          'X**J'   .
```

On the other hand, a unary invocation with a REAL*8 operand would be in error because the appropriate declaration was not made.

Operators in FLEX are restricted to a maximum of nine operands; that is, only one digit is needed to describe the position of an operand in the list of mode names.

If the operator is translated into a series of FORTRAN statements only (that is, it produces no text which is subsequently used in an expression), a resulting mode is not specified. However, a template for the translation must always be provided. If the template is too long to be specified on a single card, a sequence of several literals can be used. (Recall that a literal cannot be continued past the card boundary.)

```
          MODE VECT : REAL(25);
          OPERATOR := (VECT,VECT) =
                        ('DO &S1 I=1,25;'
                      '&S1 &1(I) = &2(I);'),
                    - (VECT) = VECT
                        ('DO &S1 I=1,25;'
                      '&S1 &0(I) = -&1(I);'
                        '&0');
```

In the preceeding example, the := operator has no resulting mode because no expression is produced to take part in succeeding operator invocations. However, a temporary variable (&0) of mode VECT is produced as the result of the - operator.

Because the translated code is not checked by FLEX to determine if it is valid FORTRAN code, an incorrect template could affect large parts of the object program. All lines of the template are assumed to be valid FORTRAN declaration or executable statements. Thus the user must not code statements like SUBROUTINE, FUNCTION, END, or comment in the template.

3.5.4.2 Function Declaration

All functions and subroutines (see next section) that are used in a program must be declared as such in a global OPERATOR statement (or several local statements if the user prefers). The information which the function declaration conveys is very similar to that provided by an operator declaration. The latter is used to define or extend an operator, while the former is used to provide mode information about a user-defined function. A function declaration is used by the processor in translating invocations of the function. Because each procedure is completely independent of all others, this information is not known to the other procedures except through a (global) function declaration.

All FLEX function invocations are translated into the standard notation that is used for FORTRAN function invocations. The user may think of a function declaration as having an implicit template associated with it and thus no template is required. However, the resulting mode of the function must be specified.

During translation of an expression, a function call is treated as if it were an operator invocation; that is, the list of mode names in the declaration is compared to the mode of the arguments in the call, and if the arguments have been incorrectly specified, no match for the (function) invocation is found. Therefore, only one declaration should be made for a particular function.

The following program segment illustrates how the function declaration is used:

```
OPERATOR FCN(REAL,INTEGER,REAL) = REAL;
PROGRAM;
    REAL A,B,C;
    INTEGER I;
    A := A + FCN(B,I,C)
ENDPROC;
```

```
FUNCTION REAL FCN(A,B,C);
    REAL A,C;
    INTEGER B;
    .
    .
    .
ENDPROC
```

The function call in the mainline is translated to produce the REAL result:

```
'FCN(B,I,C)'   .
```

3.5.4.3 Subroutine Declaration

A subroutine declaration has somewhat the same purpose as a function declaration: to describe the modes of its arguments. A single global declaration of the subroutine will be known to every procedure in the program. Subroutine calls are translated into FORTRAN CALL statements using an implicit template in the same manner as are function calls. However, since there is no resulting mode, neither a template nor a resulting mode are specified in a subroutine declaration. Only one declaration should be made for each procedure (subroutine or entry) name. The reader should consult the discussion of the CALL statement for a complete description of subroutine calls.

3.5.5 Declaration Statement

The purpose of a declaration statement is to declare variables of specific modes. Because FLEX does not have default declarations as FORTRAN does, every identifier which is used as a variable in a procedure must be declared explicitly; that is, its name must appear in some declaration statement in the procedure.

The declaration statement in FLEX has the following syntax:

```
declstmt = modename decl-list

decl = vardecl dimelt-option

vardecl = varname submode-option

varname = identifier

dimelt = (dim-list)

dim = integer
```

It is very similar to the type statement of FORTRAN. For example, consider the following declaration:

```
INTEGER*4 A,B,C,D*M   .
```

With the exception of the second submode, it looks like a FORTRAN declaration.

Because a submode can be associated with each variable name, as well as with the mode name itself, variables of several different modes can be declared in a single statement, as follows:

```
INTEGER*4 I,J*2,K,L*M(10,5)    .
```

Every variable which lacks a submode is declared to be of the mode specified initially. In the example above, I and K would be INTEGER*4, while J would be INTEGER*2 and L would be INTEGER*M.

3.5.6 Predefined Modes and Operators

The standard FORTRAN modes and operators have been predefined in FLEX. In addition, the mode STRING is permitted in order that the user may declare operators or functions whose operands are literals. However, since FORTRAN has no character string variables, the user cannot declare FLEX variables of mode STRING (that is, STRING is not a fundamental mode). The modulus operator (//) has also been defined. For a full description, the reader should consult Appendix 2.

These definitions have been included so that the user does not have to define them himself. This is very convenient since the definitions are quite lengthy. They are known to the user's entire source program; that is, they appear in the global definition area associated with the user's program.

3.6 Other Statements

3.6.1 Expressions

Expressions in FLEX are more general than those allowed in FORTRAN.

```
expr = primary
       expr operator primary

primary = opnd
          operator primary
          fcncall

opnd = var
       literal
       constant
       (expr-list)
       fcname

var = varname
      var (subscript-list)

varname = identifier

subscript = expr

fcname = identifier

fcncall = fcname (arg-list)

arg = expr
```

There are several differences between FLEX expressions and FORTRAN expressions.

The first difference is that infix notation has been generalized (see 2.3.6). Although this feature was introduced to permit operator invocation when more than two operands are specified, it is valid for binary operators as well. Thus, while one would normally code:

```
A+B
```

the invocation

```
+(A,B)
```

is also valid, if the user wishes to emphasize the similarity between operators and functions. An infix operator (that is, one which is written in standard infix notation) requires a priority in order to obtain the correct order of invocation. Prefix operators need no priorities since they are always invoked before infix operators. Thus:

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

is equivalent to the following parenthesized expression:

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

Another difference from FORTRAN is the ability to specify multiple subscript lists for the purpose of making array mode relationships clearer (see 2.3.9).

In FLEX, each operand in an expression can have an arbitrary number of prefix operators without requiring parentheses (see 2.3.7). In FORTRAN, one would code:

```
A+(-B)
```

while in FLEX, the expression:

```
A+-B
```

would be acceptable. Similarly:

```
D++(A,B)
```

and

```
D+-+(A,B)
```

are valid FLEX expressions, although they would normally be written:

```
D+(A+B)
```

and

```
D+(-(A+B))
```

respectively. The first "+" in each case is an infix operator.

A function name may be used alone only within the corresponding function procedure (see 3.4.3); it is then treated essentially like a variable name. In the other procedures the function name indicates an invocation of the function. This is exactly how a function name is used in FORTRAN.

The assignment operator in FLEX is ":=" . Normally it appears in only two contexts:

a) as the leftmost operator in an assignment statement, and

b) in the DO statement.

It translates into an ordinary FORTRAN assignment statement. However, if the user wishes to make appropriate definitions, he may embed the := operator in an expression.

3.6.2 Assignment Statement

In FLEX, the assignment statement has the following syntax:

```
assignstmt = var := expr
             var = expr

var = varname
      var (subscript-list)

varname = identifier

subscript = expr
```

As in FORTRAN, the purpose of an assignment statement is to assign the value of the expression to the (subscripted) variable appearing on the left side of the assignment operator.

Any imperative which does not start with either a keyword or a mode name is assumed to be an assignment statement. The first operator (with the exception of those occurring within parentheses) may be either

= or := . If the = operator is used, it will be treated as the assignment operator; that is, one of the templates corresponding to the := operator will be used for producing the translation. Every other occurrence of the = operator remains unchanged. This permits the user to write FORTRAN-like assignment statements using the = operator instead of the assignment (:=) operator. Thus:

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

are both translated into the same code.

3.6.3 Input/Output Statements

There are four input/output statements in FLEX: READ and WRITE for formatted I/O, and INPUT and OUTPUT for unformatted record I/O. The latter two statements are used less frequently. The I/O statements in FLEX have the following syntax:

```
iostmt = ioname (unitno) ioitem-list
         ioname, ioitem-list

ioname = READ
         WRITE
         INPUT
         OUTPUT

unitno = expr

ioitem = expr-option : fmtitem
         expr
         loopitem

fmtitem = fmt
          (fmt-list)

fmt = fmtname exprelt-option
      controlname exprelt-option

exprelt = (expr-list)

loopitem = DO (initclause controlclause, ioitem-list)
```

```
fmtname = D
          E
          F
          I
          L

controlname = BLANK
              COLUMN
              PAGE
              SKIP
```

As discussed in Chapter 2, the omission of the unit number implies continuation of a previous I/O statement. This permits the user to intersperse other statements among the different portions of an I/O operation. Thus, it is similar to STREAM I/O in PL/1. For example:

```
WRITE(6) X:E(10,4),:BLANK(10);
IF F(X)¬=0
   THEN WRITE, G(X)/F(X):E(10,4)
   ELSE WRITE, 'UNDEFINED'
FI;
```

There are five different format names recognized by FLEX; the FORTRAN names have been adopted for simplicity. They control the reading and writing of the fundamental modes as follows:

a) E and F are used for REAL*4 values,

b) D and F are used for REAL*8 values,

c) I is used for both INTEGER*2 and INTEGER*4 values, and

d) L is used for both LOGICAL*1 and LOGICAL*4 values.

When reading or writing values of a new mode, the format name corresponding to the related fundamental mode should be used. As in FORTRAN, I and L format specifications require only a single value, the field width. D, E and F specifications may have a second value giving the number of decimal positions. In all five format specifications, a zero or negative field width will result in the omission of the field (and therefore the data item). When no format specification is given for a particular data item, default values are supplied according to the mode of the item. The default values are given in Appendix 2.

The four control functions are used to control the position at which the next read or write is to occur. They are:

a) BLANK, which causes the specified number of columns to be skipped in the input record, or blanks inserted in the output record,

b) COLUMN, which causes the next read or print to start at the column specified,

c) PAGE, which causes the next printer output to begin at the top of a new page, and

d) SKIP, which causes the specified number of lines to be skipped.

Each control function may have one argument; if no argument is supplied, the value 1 is assumed. If the value of the argument is zero or negative, the control item is omitted, with the exception of SKIP(0) which causes re-positioning to the beginning of the current record; that is, SKIP(0) is equivalent to COLUMN(1).

There are several ways in which items can be specified in an I/O statement. For example, if a REAL variable A is to be printed starting at column 20 using default format, the statement would be coded in one of the following ways:

```
WRITE(6) A:COLUMN(20 );

WRITE(6)  :COLUMN(20),A;
```

If it were to have a field width of 15, with 5 decimal places, the statement would be either:

```
        WRITE(6) A:(COLUMN(20),E(15,5));
or      WRITE(6) :COLUMN(20), A:E(15,5);
```

Loops can be used in I/O statements just as in FORTRAN. However, the syntax of a loop is different than FORTRAN syntax. The loop control information appears at the beginning and resembles the FLEX DO clause syntax. For example, if B were a dimensioned REAL variable with N values to be read, the statement might look like:

```
READ(5)N, DO(I=1 TO N, B(I));
```

The loop can contain a list of data or control items or even other loops. For example:

```
READ(5) DO(I=1 TO 15, B(I),
             DO(J=1 TO 7, A(I,J), C(J,I)));
```

3.6.4 CALL Statement

The FLEX CALL statement is just like the FORTRAN CALL statement. It has the following syntax:

```
callstmt = CALL procname (arg-list)
procname = identifier
arg = expr
```

and its purpose is to transfer control to one of the entry points in a subroutine procedure, as well as to establish a one-to-one correspondence between the parameters in the SUBROUTINE or ENTRY statement and the arguments in the CALL statement. The point to which control is transferred is determined by the procedure name specified in the call. The number of arguments must be the same as the number of parameters expected at that entry point, and their modes must be the same. If the following subroutine has been defined:

```
          SUBROUTINE TEST(A,B,C,D);
          REAL A,D;
          VECT B;
          INTEGER C;
          .
          .
          .
          D = ...,
          ENDPROC
```

and the appropriate subroutine declaration:

```
          OPERATOR TEST(REAL,VECT,INTEGER,REAL);
```

has been made, then the subroutine would be called as follows:

```
          REAL X,Y;
          VECT Z;
          INTEGER I,J;
          .
          .
          .
          CALL TEST(X*Y,X,I-J,X);
```

The correspondence between parameters of an entry point and arguments of a procedure invocation is established exactly the same way that it is in FORTRAN. However, FLEX does not permit arguments to be passed by value as in FORTRAN; all arguments are passed by address.

3.6.5 STOP Statement

The function of the STOP statement is to cause the program to terminate execution. It has the following syntax:

```
stopstmt = STOP
```

This statement may appear anywhere that any of the other statements are allowed.

The STOP statement is not the only way to cause execution to halt. Execution also ends when the ENDPROC in the mainline is encountered. Thus the STOP statement usually need not be coded unless execution is to terminate at a place other than the end of the mainline procedure.

3.6.6 RETURN Statement

The function of the RETURN statement is to cause control to be transferred back to the procedure which called the current procedure. It has the following syntax:

```
returnstmt = RETURN
```

The RETURN statement may be coded in the mainline, function or subroutine procedures. In the first case, it is treated exactly like a STOP statement; that is, the program terminates. In the other cases, control returns to the calling procedure. Control also returns to the calling procedure whenever the ENDPROC of a function or subroutine procedure is encountered. This means that the RETURN statement may be omitted whenever the transfer of control takes place at the end of a function or subroutine.

3.6.7 GOTO Statement

The purpose of the GOTO statement is to cause a transfer of control to the imperative having the label specified in the GOTO statement. Its syntax is as follows:

```
gotostmt = GOTO label
label = identifier
```

Labels are translated into FORTRAN statement numbers. Whenever the label is referenced (in a GOTO statement), it is replaced by the appropriate statement number. Thus the following piece of code:

```
          GOTO SKIP;
          .
          .
          .
      SKIP: A:=B+C;
```

might be translated as follows:

```
          GOTO 10
          .
          .
          .
      10  A=B+C
```

Of course, in a structured program, the GOTO statement would be used quite infrequently.

3.7 COMMENT Statement

The COMMENT statement provides the user with the ability to write an explanatory comment that crosses card boundaries. This is not allowed when card comments are used (see 3.2.3). In general, programs are introduced with a long comment explaining the purpose of the program and introducing the main variables, where necessary. The COMMENT statement is provided specifically for this type of comment. The short comments which generally appear within a program can be written using the card comment.

The COMMENT statement begins with the keyword COMMENT and ends with a semi-colon. Between the COMMENT and the semi-colon may appear any sequence of symbols, with the exception of the semi-colon.

If the user omits the ending semi-colon part of his program may be included in the COMMENT but there will be no error message. The user should verify that statements of his program are numbered (Appendix 3).

3.8 Additional Features

Several features in FLEX have not yet been described. They are used somewhat infrequently, and have been gathered here rather than being included with the basic concepts.

It has been mentioned several times that the standard FORTRAN modes, operators and priorities have been predefined. This does not mean that these definitions cannot be changed. The user may redefine any of them by using the definition statements. For example, the priority of the operator + could be redefined as follows:

```
PRIORITY + = * ;
```

An operator or mode which is redefined within a procedure will be in effect only during that procedure. Its original definition will be restored in the other procedures. The user may redefine any mode, operator or priority which he has added in previous definition statements; however these redefinitions are effective for the remainder of the program and are not reset at the end of the procedure.

In the description of function and subroutine calls given so far, prefix notation has always been used. Because these invocations are treated just like operator invocations, the generalized form of infix notation can be used. Thus a function OF may be called as follows:

```
F OF X
```

rather than using the usual prefix notation:

```
OF(F,X) .
```

However, the user should recall that all infix operators have priorities so that they can be invoked in the proper order. Thus, any function which is invoked using infix notation must also have a priority. In general, this priority should be higher than that of every other operator.

A third feature that has not yet been described is giving a list of mode names in place of a single mode in the OPERATOR statement. For example, the definition of scalar multiplication given in Appendix 4 is:

```
          OPERATOR * ((REAL,INTEGER),POLY) = POLY
                     ('DO &S1 I$=1,16;'
                      '&S1 &0(I$)=&1*&2(I$);'
                      '&0');
```

For this definition to be selected, the first operand may be either REAL or INTEGER and the second must be POLY. The user should exercise care when using this feature. Any operator invocation which translates into a FORTRAN subroutine or function call requires a unique mode for each operand, and thus the specification of a list of modes will lead to execution-time errors. This is also the case for any function or subroutine declaration. Each operand must have a unique mode. However, in cases like the one given above, several definitions can be combined into one by using the multiple mode specification for a single operand. This, of course, depends on the fact that FORTRAN multiplication of a REAL by either a REAL or an INTEGER has the same code.

The complete syntax of FLEX is given in Appendix 1. Built-in definitions are given in Appendix 2 and two sample FLEX programs are given in Appendix 4.

Chapter 4

CONCLUSIONS

There were two original design goals for FLEX, but as the implementation progressed, other improvements and additions were incorporated. The result is a language which is largely successful in accomplishing these goals. As discussed below, there are several major and many minor improvements which can be made in FLEX.

The first design goal was to permit the programmer to define new modes and operators, and it arose from the desire for a convenient and readable way to perform multiple precision computations. In this respect, FLEX has made it very easy to define and use multi-precise variables in a program. The general way in which these definitions are implemented permits the programmer to add arbitrary data types. That the object code is FORTRAN, however, precludes the definition of complicated data structures.

The second goal was extension of FORTRAN to include several reasonably powerful control structures. The structures included in FLEX are certainly a great deal more convenient than those in FORTRAN. In some respects, they are better than those in PL/1. For example, PL/1 does not have a CASE statement, a legitimate and very useful construct. The simple DO group has been eliminated by having an IF construct very similar to that in ALGOL 68 [4]. Again, the structure of FORTRAN limits the control structures that may be implemented; for example, nesting of procedures is not permitted.

One of the most important improvements was the extension of the input/output statements. The FLEX I/O statements incorporate several changes from those in FORTRAN. For example, format specifications are included in the I/O statement itself, rather than in a separate FORMAT statement as in FORTRAN. In addition, expressions are allowed in the statements, thus making them more general and more convenient for the programmer.

The following paragraphs describe improvements which have not yet been implemented.

Changes to the handling of input and output are of major importance. The present method of specifying I/O routine names in the MODE statement is not very satisfactory. It would be more consistent to declare them very much like operators in the OPERATOR statement, since input and output are really just operations on the data items which appear in the I/O statements. This would have several advantages. It would permit more complex code than a simple subroutine call to be generated for the input and output of values. If the format specifications were also included as possible arguments, then the user would be able to make several definitions for each mode depending on the type and number of format specifications:

```
OPERATOR READ(INTEGER*4) = ('CALL IOFREE(&1)');
         READ(INTEGER*4,INTEGER*4) =
              ('CALL IO$I4F(&1,&2)');
```

At the present time, hexadecimal (Z) and alphameric (A) input/output is not allowed for any data types. Each is read or written by a single routine as described in Appendix 2.

In order to simplify some of the internal organization of FLEX and to eliminate the need for lists of modes in the OPERATOR statement, it would be convenient if FLEX would automatically convert values from one data type to another (for example, INTEGER to REAL to COMPLEX). This is similar to coercions in ALGOL 68. Declaration of the desired conversions would be made in the MODE statement.

The multiple assignment statement would permit several computations to be made before the values were assigned to the variables on the left hand side of the assignment operator. For example, in the following statement:

```
(X,Y) := (X+1, X*Y)
```

once the two computations X+1 and X*Y were done, the values would be assigned to X and Y, respectively. The effect is that the old value of X is used in the second computation. Another very common situation in which this construct would be useful is:

```
(X,Y) := (Y,X)
```

which interchanges the values of X and Y . In general, the assignment of values to variables would be one-to-one, but the same expression could be assigned to all or some of the variables according to certain rules. For example, if fewer expressions than variables were specified, then the last expression would be assigned to the remaining variables.

Several statements in FORTRAN have not yet been implemented in FLEX because they were considered relatively unimportant, perhaps even undesirable. They are:

```
IMPLICIT
EQUIVALENCE
COMMON
```

Their syntax would be very similar to that of the corresponding FORTRAN statements.

Initialization of variables in declaration statements has not yet been implemented. In FLEX, initialization will be permitted in the MODE statement as well. This will permit the automatic initialization of all variables of that mode, thus eliminating the need for individual initialization.

A cross reference table is a useful debugging tool. Provision has been made in FLEX to implement such a feature in order to give the programmer a list of every reference to the variable names, modes and operators used in his program.

FLEX has not yet been used extensively and, therefore, a complete assessment of its language features is not possible. On the basis of the programs that have been run, however, it has proven itself a far superior language to FORTRAN, and sometimes to PL/1 (if the programmer does not need, or cannot cope with, the esoteric features of PL/1). It has certainly fulfilled its original goal of providing easy access to multiple precision computations, and it is hoped that it may be used successfully for other similar applications.

APPENDIX 1

FLEX Syntax

This appendix contains the complete syntax description of the current version of FLEX.

In the source program, the following pairs of words are synonyms; that is, they may be used interchangeably:

ENDIF	FI
ENDCASE	ESAC
GOTO	GO TO

Only the first member of each pair appears in the syntax.

The two operators, := and =, are different. However, in the DO statement and the assignment statement, the = operator may be used where the assignment operator is intended. When = is used in these contexts, it is translated into the assignment operator. This permits the = operator to be used as it is in FORTRAN.

```
sourcepgm = globaldefn; proc-series
            proc-series

globaldefn = defnstmt-series

defnstmt-series = defnstmt
                  defnstmt-series; defnstmt

defnstmt = modestmt
           priostmt
           operstmt
           cmtstmt
proc-series = proc
              proc-series; proc

proc = pgmstruc
       fcnstruc
       subrstruc

pgmstruc = pgmstmt; procstmt-series ENDPROC

fcnstruc = fcnstmt; procstmt-series ENDPROC

subrstruc = subrstmt; subrstmt-series ENDPROC

procstmt-series = procstmt
                  procstmt-series; procstmt

subrstmt-series = subrstmt
                  subrstmt-series; subrstmt

procstmt = labeldefn-sequence imperative
           imperative
           defnstmt
           declstmt

subrstmt = procstmt
           entrystmt

labeldefn-sequence = labeldefn
                     labeldefn-sequence labeldefn

labeldefn = label:

imperative = loop
             selection
             stmt

loop = dostruc
       whilestruc

selection = casestruc
            ifstruc
```

```
stmt = assignstmt
       iostmt
       callstmt
       stopstmt
       rêturnstmt
       gotostmt


modestmt = MODE modelt-list

modelt-list = modelt
              modelt-list, modelt

modelt = modedefn
         modedefn ioelt-sequence

modedefn = modename : oldmode
           modename = oldmode

oldmode = modename
          modename dimelt

ioelt-sequence = ioelt
                 ioelt-sequence ioelt

ioelt = ioname (iortn)

iortn = identifier

priostmt = PRIORITY prioelt-list

prioelt-list = prioelt
               prioelt-list, prioelt

prioelt = operator = prio

prio = integer
       operator

operstmt = OPERATOR operelt-list

operelt-list = operelt
               operelt-list, operelt

operelt = operdecl
          fcndecl
          subrdecl

operdecl = operator (mode-list) = resultmode template
           operator (mode-list) = template

fcndecl = fcname (mode-list) = resultmode
```

```
subrdecl = procname (mode-list)

mode-list = mode
            mode-list, mode

mode = modename
       (modename-list)

modename-list = modename
                modename-list, modename

template = (literal-sequence)

literal-sequence = literal
                   literal-sequence literal

pgmstmt = PROGRAM

fcnstmt = FUNCTION resultmode fcname (parm-list)

subrstmt = SUBROUTINE procname (parm-list)

declstmt = modename decl-list

decl-list = decl
            decl-list, decl

decl = vardecl
       vardecl dimelt

vardecl = varname
          varname submode

entrystmt = ENTRY procname (parm-list)

dostruc = dostmt END
          dostmt; procstmt-series END

whilestruc = whilestmt; procstmt-series END

dostmt = DO initclause controlclause

initclause = varname := expr
             varname = expr

controlclause = testclause
                whileclause
                testclause byclause
                testclause whileclause
                byclause whileclause
                testclause byclause whileclause
testclause = TO expr
             DOWNTO expr

byclause = BY expr

whileclause = WHILE expr

whilestmt = WHILE expr

casestruc = CASE caseclause ENDCASE

ifstruc = IF ifclause ENDIF

caseclause = expr inclause
             expr inclause orclause-sequence
             expr inclause outclause
             expr inclause orclause-sequence outclause

inclause = IN procstmt-series

orclause-sequence = orclause
                    orclause-sequence orclause

orclause = OR procstmt-series

outclause = OUT procstmt-series
            OUTCASE caseclause

ifclause = expr thenclause
           expr thenclause elseclause

thenclause = THEN procstmt-series

elseclause = ELSE procstmt-series
             ELSEIF ifclause

gotostmt = GOTO label

callstmt = CALL procname (arg-list)

stopstmt = STOP

returnstmt = RETURN

assignstmt = var := expr
             var = expr

iostmt = ioname, ioitem-list
         ioname (unitno) ioitem-list

unitno = expr
```

```
ioitem-list = ioitem
              ioitem-list, ioitem

ioitem = expr
         expr : fmtitem
         : fmtitem
         loopitem

fmtitem = fmt
          (fmt-list)

fmt-list = fmt
           fmt-list, fmt

fmt = fmtname
      fmtname (expr-list)
      controlname
      controlname (expr-list)

loopitem = DO (initclause controlclause, ioitem-list)

fmtname = A
          D
          E
          F
          I
          L
          Z

controlname = BLANK
              COLUMN
              PAGE
              SKIP

cmtstmt = COMMENT charstringexclsemicolon

modename = identifier
           identifier submode

submode = * identifier
          * integer

expr = primary
       expr operator primary

primary = opnd
          operator primary
          fcncall

opnd = var
       literal
       constant
       (expr-list)
       fcname
```

```
var = varname
      var (subscript-list)

varname = identifier

subscript-list = subscript
                 subscript-list, subscript

subscript = expr

expr-list = expr
            expr-list, expr

fcname = identifier

fcncall = fcname (arg-list)

arg-list = arg
           arg-list, arg

arg = expr

resultmode = modename

procname = identifier

label = identifier

parm-list = parm
            parm-list, parm

parm = identifer

dimelt = (dim-list)

dim-list = dim
           dim-list, dim

dim = integer

ioname = INPUT
         OUTPUT
         READ
         WRITE

identifier = letter
             identifier letter
             identifier digit

literal = 'charstring'
          "charstring"

integer = digit
          integer digit
```

```
operator = identifier
           specialcharacter
           <=
           ||
           **
           ¬<
           ¬>
           ¬=
           ->
           //
           >=
           :=

empty =

constant = integerconstant
           shortintegerconstant
           realconstant
           longrealconstant
           imaginconstant
           longimaginconstant
           logicalconstant

integerconstant = sign integer

sign = empty
       +
       -

shortintegerconstant = integerconstant S

realconstant = decimalconstant
               decimalconstant realexponent
               integerconstant realexponent

decimalconstant = integerconstant .
                  integerconstant . integer
                  sign . integer

realexponent = E integerconstant

longrealconstant = decimalconstant longrealexponent
                   integerconstant longrealexponent

longrealexponent = D integerconstant

imaginconstant = integerconstant I
                 realconstant I

longimaginconstant = longrealconstant I

logicalconstant = TRUE
                  FALSE
```

APPENDIX 2

Predefined Modes and Operators

This appendix discusses the definitions which have been made so that ordinary FORTRAN data types and expressions may be written in FLEX much as they are in FORTRAN.

Modes

There are eight fundamental modes which exist in FORTRAN, and which have been defined in FLEX. They are:

INTEGER*2	(I2)
INTEGER*4	(I4)
REAL*4	(R4)
REAL*8	(R8)
COMPLEX*8	(C8)
COMPLEX*16	(C16)
LOGICAL*4	(L4)
LOGICAL*1	(L1)

The names in parentheses will be used to refer to these modes in the discussion and tables that follow. The first six are referred to as arithmetic modes and the last two as logical modes.

The following mode equivalences have also been predefined:

```
MODE INTEGER = INTEGER*4,
        REAL = REAL*4,
     COMPLEX = COMPLEX*4,
     LOGICAL = LOGICAL*4;
```

In addition, mode STRING has been defined; it is the mode of literals. The user cannot declare variables of mode STRING because of restrictions in FORTRÅN.

Constants

With the exception of L1, there are constants of every fundamental mode. Integer (I4), real (R4) and long real (R8) constants are written just as they are in FORTRAN. Short integer (I2) constants are just integer constants with the letter S at the end (with no intervening characters). Imaginary (C8) and long imaginary (C16) constants are designated by a (long) real constant followed by the letter I . Thus complex constants must be written in two parts, as in:

1.2 + 3.4I .

Logical (L4) constants are designated by TRUE and FALSE (without periods). A complete description of these constants may be found in Appendix 1.

FLEX imposes no restrictions on the size of an arithmetic constant. However, for the program to execute correctly, constants should conform to the size restrictions imposed by the FORTRAN compiler which is used to compile the object program.

Priorities

The operators have been assigned priorities so that the usual FORTRAN order of operations is preserved. The actual priorities are:

:=	100
|	200
&	300
< <= > >= = ¬= ¬< ¬>	500

+ -	600
* / //	700
**	800

Operators written in prefix notation are always invoked first; that is, they effectively have a higher priority than any operator written in infix notation. All unary operators are written using prefix notation, whereas binary operators are usually written in infix notation. However, they may be written in prefix notation if desired. In this case, the priorities listed above are irrelevant.

Operator Definitions

Rather than writing out all the operator definitions which have been predefined for use in FLEX, they are explained in the following paragraphs, using tables where applicable.

With the exception of := , all the operators have resulting modes. In the tables, the modes of the operands appear as the row and column indices, while the resulting modes appear as the entries in the tables. If no resulting mode is shown for a particular combination of operand modes, that combination has not been predefined. If the same table applies to more than one operator, the table is given only once, with a list of the operators to which it applies.

The templates used in translating the operator invocations are also given. In general, the same template is used for every combination of modes that is defined for a particular operator. In this case, the template is specified only once, with the operator to which it refers.

1) Binary := '&1=&2;'

All mode combinations such that both operands are arithmetic or both are logical are defined for the assignment operator := . There is no resulting mode; invocation of the assignment operator simply produces a FORTRAN assignment statement.

2) Binary + '&1+&2'

- '&1-&2'

* '&1*&2'

/ '&1/&2'

&1 \ &2	I2	I4	R4	R8	C8	C16
I2	I4	I4	R4	R8	C8	C16
I4	I4	I4	R4	R8	C8	C16
R4	R4	R4	R4	R8	C8	C16
R8	R8	R8	R8	R8	C8	C16
C8	C8	C8	C8	C8	C8	C16
C16	C16	C16	C16	C16	C16	C16

3)	Binary	<	'&1.LT.&2'
		<= ¬>	'&1.LE.&2'
		>	'&1.GT.&2'
		>= ¬<	'&1.GE.&2'
		=	'&1.EQ.&2'
		¬=	'&1.NE.&2'

The comparison operators are defined between any pair of the modes I2, I4, R4 and R8. The resulting mode is always L4.

4)	Binary	&	'&1.AND.&2'
		|	'&1.OR.&2'

The logical operators are defined between any pair of the logical modes. The result is always L4.

5)	Unary	¬	'.NOT.&1'

The operand may be either L1 or L4 but the result is always L4.

6)	Unary	+	'+&1'
		-	'-&1'

If the operand is of mode I2, the result is I4 ; otherwise the resulting mode is the same as the (arithmetic) mode of the operand.

7) Binary   **   '&1**&2'

&2 / &1	I2	I4	R4	R8
I2	I4	I4	R4	R8
I4	I4	I4	R4	R8
R4	R4	R4	R4	R8
R8	R8	R8	R8	R8
C8	C8	C8		
C16	C16	C16		

8) Binary   //

```
OPERATOR // (INTEGER*4,INTEGER*4) = INTEGER*4 ('MOD(&1,&2)'),
         // (REAL*4,REAL*4) = REAL*4 ('AMOD(&1,&2)'),
         // (REAL*8,REAL*8) = REAL*8 ('DMOD(&1,&2)');
```

Input and Output

The fundamental modes of FORTRAN already have I/O routines associated with them [7]. These routines provide an interface between the FLEX object code and the FORTRAN execution-time format routines. Each routine has at least one argument: the expression being printed or the variable being read. This argument may be followed by up to three others: the field width, the number of decimal places and the scale factor. The latter two are used only with real and complex values. If only the expression (or variable) is specified, free format is assumed, a description of which is given in Chapter 2. To output a value under free format, default values for the field width, and so on, are supplied by the routines. These default values are given in the table given below.

mode	routine	field width	no. of dec. pl.	scale
L1	IO$L1	4		
L4	IO$L4	12		
I2	IO$I2	8		
I4	IO$I4	13		
R4	IO$R4	16	7	0
R8	IO$R8	23	14	0
C8	IO$C8	16	7	0
C16	IO$C16	23	14	0

In the case of complex expressions, two real values are printed, using the same format for each.

APPENDIX 3

Compilation and Execution of FLEX Programs

A sample Job Control Language (JCL) listing, which was used to compile and execute a FLEX program, is given at the end of this appendix. As shown in the example, five files are required by the FLEX step. They are:

a) STEPLIB, a partitioned data set (library) containing FLEX

b) TRANSIN, containing the FLEX source program

c) TRANSOUT, for the FLEX program listing

d) SYSPUNCH, for the object program

e) SYSUT1, for temporary use during the compilation.

The SYSUT1 file is used to hold the object statements until the end of the procedure when the declaration statements of that procedure are written into the SYSPUNCH file. The SYSUT1 file is then copied to the SYSPUNCH file following the declarations. This permits declarations to be made throughout the source program if the user wishes; in the object code all the declarations are moved to the beginning of the procedure.

The FORTRAN SYSPRINT file (FLEX object code) is usually suppressed. However, if the user wishes to look at it, the DD-card in the example may be removed.

Source Listing

A sample listing of a FLEX source program is given at the end of this appendix. Every input record is numbered sequentially and this number is used to identify each record uniquely. In addition, a second number appears on some records. It is the number of the first new statement which starts on that record. The records are printed exactly as they were read.

If errors are found in a record, error messages are produced in the listing; for most errors a pointer ($) is used to indicate the position where the error occurred in the record. Some messages indicate errors on preceding records, these causing no pointers to be produced; however, the appropriate record and statement numbers are given in the messages.

Messages and Completion Codes

All messages produced by FLEX begin with:

STATEMENT XXXXX, LINE YYYYY

where XXXXX is the number of the FLEX statement being processed and YYYYY is the number of the input record on which the error was found. Most messages precede the record to which they refer. When they do, the position of the error is marked by a "$" printed directly above it.

Messages in FLEX indicate one of the four types of errors described below.

a) Fatal errors cause compilation to be abandoned and control to be passed back to the system immediately (condition code is 16).

b) Severe errors usually cause text to be omitted (condition code is 12).

c) Errors do not cause text omission (condition code is 8).

d) Warnings indicate a possible error situation although correct object code is produced (condition code is 4).

Both errors and severe errors cause production of object code to be inhibited for the remainder of the statement. In additon, FLEX generates a call to a run-time error routine, E$STP, so that, if the object program is actually run (and it should not be), an execution time error will be produced. In general, the FORTRAN step is run if the condition code from the FLEX step is not greater than 8 (as shown in the JCL listing). Thus, if a severe or fatal error is found, the FORTRAN step is bypassed.

Fatal Errors

1. STATEMENT STACK OVERFLOW

Nesting of control structures is too deep.

2. GLOBAL DEFINITION STACK OVERFLOW

A stack used when local definitions are being released at the end of a procedure has overflowed.

3. ARITHMETIC STACK INCORRECTLY BUILT

This message indicates a compiler error.

4. SYMBOL TABLE OVERFLOW

Too many variables, operators and modes have been declared in the program.

Severe Errors

1. ARITHMETIC STACK OVERFLOW. STATEMENT ABANDONED.

An expression is too complex and/or control structures are nested too deeply.

2. PRECEDING STATEMENT SHOULD END WITH SEMI-COLON

Compilation resumes at the next recognizable statement; that is, after the next semi-colon or punctuation word is found, or when a keyword is found.

3. NAME PREVIOUSLY DEFINED - OLD DEFINITION USED

The variable being declared has already been declared, or the name is a keyword, punctuation word, etc.

4. ILLEGAL MODE NAME

The mode name is not an identifier, or has been defined as something other than a mode.

5. INVALID SUBMODE

The submode is neither an identifier nor an integer.

6. OLDMODE CLAUSE IS MISSING

The oldmode clause has been omitted, or an invalid symbol has been used for the relation.

7. ILLEGAL OPERATOR SPECIFIED

It must be an identifier, a special character or one of the special character pairs.

8. EQUAL SIGN EXPECTED

The PRIORITY statement is coded incorrectly.

9. MODE LIST NOT SUPPLIED

The list of mode names in the OPERATOR statement has been omitted or coded improperly.

10. LITERAL EXPECTED

A template has not been specified correctly.

11. ATTEMPT TO REDECLARE FUNCTION

A function name should be declared only once in a procedure, or once in the global area, but not both.

12. LABEL IDENTIFIER EXPECTED - STATEMENT ABANDONED

The label in a GOTO statement is not an identifier.

13. VARIABLE WAS EXPECTED AS DO INDEX

The DO index variable should follow the DO immediately (or the left parenthesis in an I/O loop).

14. MODE OF DO INDEX IS UNDEFINED

The DO index variable must be declared.

15. I/O ROUTINE NAME IS MISSING IN MODE STATEMENT

The routine name should be enclosed in parentheses directly following the I/O keyword.

16. THIS KEYWORD SHOULD NOT APPEAR IN I/O STATEMENT

Only the keyword DO is valid within the I/O statements.

17. MISSING RIGHT PARENTHESIS IN I/O LOOP

The right parenthesis at the end of an I/O loop is omitted.

18. EXCESSIVE NUMBER OF DIMENSIONS - STACK OVERFLOW

The number of dimensions declared for a variable is too large to be stored in the stack area.

19. DIMENSION STRING TOO LONG - OVERFLOW

The dimension list for a variable being declared in the object code has overflowed the area in which it is being constructed.

20. STACK OVERFLOW - TOO MANY LEVELS OF PARENTHESES IN TRANSLATION

The translation of an operator invocation contains too many levels of parentheses or too many nested subscripts.

21. IDENTIFIER WAS EXPECTED

An identifier is required in a declaration statement.

22. RIGHT PARENTHESIS EXPECTED

No right parenthesis exists to match a preceding left parenthesis.

23. UNMATCHED QUOTE

A literal may not cross a card boundary.

Errors

1. MISSING PROGRAM STATEMENT

The user has placed non-global statements in the global definition area, or has omitted the PROGRAM statement.

2. INVALID FUNCTION NAME

The name of a function is not an identifier.

3. PARAMETER LIST IS MISSING

Every function must have at least one parameter.

4. FUNCTION NAME DECLARED PREVIOUSLY

Name being declared as a function was previously declared as an operator or subroutine.

5. INVALID PROCEDURE NAME

The name of a SUBROUTINE or ENTRY point is not an identifier.

6. PARAMETER NAME HAS BEEN USED PREVIOUSLY

A parameter name should not be declared or used before it appears in a parameter list.

7. MISSING STRUCTURE TERMINATOR

The END, ENDIF, ENDCASE or ENDPROC is missing, or the control structures are improperly nested.

8. PROCEDURE STATEMENT OR PUNCTUATION SHOULD NOT BE LABELLED

PROGRAM, FUNCTION, SUBROUTINE and ENTRY statements should not be labelled because of restrictions in FORTRAN. Labelling a punctuation word is equivalent to labelling the end of the preceding clause, not the following clause.

9. NAME OF LABEL DEFINED PREVIOUSLY

An identifier used as a label has been defined previously. The user may be attempting to use a reserved word.

10. LABEL HAS BEEN USED PREVIOUSLY - NEW DEFINITION IS USED

The user is attempting to redefine a label.

11. PUNCTUATION WORD IS ILLEGAL HERE. IT IS OMITTED.

A punctuation word has been used improperly. The control structures may be nested incorrectly.

12. PUNCTUATION WORD "THEN" WAS EXPECTED

THEN is missing or improperly specified in IF selection.

13. PUNCTUATION WORD "IN" WAS EXPECTED

IN is missing or improperly specified in CASE selection.

14. INFINITE DO LOOP HAS BEEN GENERATED

A TO clause, DOWNTO clause or WHILE clause must be specified to provide an exit from the loop.

15. MISSING "TO"

The TO in the GO TO statement is missing.

16. TRANSLATION OF THIS MODE IS NOT A FORTRAN MODE

A new mode is not related to one of the fundamental (FORTRAN) modes.

17. CIRCULAR MODE DEFINITION

The user is attempting to relate a new mode to itself, either directly or indirectly.

18. CIRCULAR PRIORITY DEFINITION - 200 ASSUMED

The user is attempting to define the priority of an operator in terms of itself, either directly or indirectly.

19. I/O ROUTINE NAME IS NOT AN IDENTIFIER

The subroutine name specified in a MODE statement is not a valid identifier.

20. COMMA SHOULD FOLLOW I/O KEYWORD

If no unit number is specified, there should be a comma after the keyword.

21. UNIT NUMBER SHOULD BE INTEGER EXPRESSION

Any expression used to identify the unit for I/O should be INTEGER*4.

22. FORMAT NAME IS UNDEFINED

The format name in an I/O statement is not one of those which is recognized by FLEX.

23. FORMAT EXPRESSION SHOULD BE INTEGER

Any expression for the field width, and so on, should be INTEGER*4.

24. NO OUTPUT EXPRESSION PROVIDED

A format item has been supplied but there is no corresponding expression for output, or variable for input.

25. I/O ROUTINE NOT SUPPLIED

An I/O routine corresponding to the I/O statement keyword has not been supplied for the mode of the output expression or for any of its related modes.

26. MISSING LEFT PARENTHESIS IN I/O LOOP

There must be a left parenthesis between the DO and the following clauses.

27. MISSING COMMA IN I/O LOOP

The DO clauses must be separated from the list of I/O items by a comma.

28. UNMATCHED RIGHT PARENTHESIS

A right parenthesis matches no preceding left parenthesis in an I/O statement.

29. DIMENSION IS NOT AN INTEGER CONSTANT

In a dimensioned variable or mode array declaration, the dimensions must be integer constants.

30. VARIABLE NAME WAS EXPECTED BEFORE SUBSCRIPTS

User is attempting to subscript a quantity other than a dimensioned variable name.

31. VARIABLE HAS NOT BEEN DECLARED

User is attempting to use an undeclared variable.

32. INCORRECT NUMBER OF SUBSCRIPTS

The number of subscripts specified does not correspond to the number in the variable declaration and/or in the array mode declarations.

33. TOO MANY SUBSCRIPTS - DELETED AT RIGHT

The user has specified more subscripts than dimensions declared for the variable and/or the array modes.

34. OPERAND EXPECTED - PUNCTUATION IS OMITTED

A comma, right parenthesis or colon has been found when an operand was expected. It is ignored.

35. OPERAND EXPECTED - DUMMY VALUE IS ASSUMED

For the purpose of compiling the rest of the expression, a dummy operand is inserted. The operand is not declared; that is, it has no mode.

36. OPERAND EXPECTED - PERIOD IS IGNORED

A period has been found when an operand was expected. The period is ignored.

37. TWO CONSECUTIVE OPERANDS - COMMA INSERTED

Two operands appear with no separating operator. They are assumed to be separated by a comma.

38. PERIOD IS AN INVALID OPERATOR - IT IS IGNORED

A period should not appear in an expression where an operator is expected.

39. UNMATCHED LEFT PARENTHESIS

A right parenthesis is missing in an expression.

40. NO TEMPLATE SUPPLIED FOR THESE MODES

An operator definition has not been made for the modes that have been used.

41. INVALID PARAMETER INDICATOR IN TEMPLATE

An "&" found but it is not &n or &Sn.

42. NUMBER OF OPERANDS IS LESS THAN PARAMETER NUMBER

The template contains a parameter reference whose number is larger than the number of operands in the definition.

43. NO DEFINITION FOUND TO MATCH THIS FUNCTION OR OPERATOR INVOCATION

A match has not been found for the given list of operands; that is, the appropriate declaration has not been made.

44. UNMATCHED RIGHT PARENTHESIS IN TRANSLATION

The user has omitted a right parenthesis when coding a template.

45. INCOMPLETE ASSIGNMENT STATEMENT, OR MISSING SEMI-COLON IN TRANSLATION TEMPLATE

The user has coded an invalid assignment statement or he has forgotten the semi-colon in the template which causes the statement to be produced in the object code file.

46. MULTIPLE PERIODS OCCUR IN THIS CONSTANT

A real or complex constant has been specified incorrectly.

47. S CANNOT BE USED WITH FLOATING POINT NUMBER

The short integer marker "S" should not be appended to a floating point number.

48. TEXT BETWEEN $'S HAS BEEN OMITTED

Severe errors often cause text to be omitted. This message indicates exactly which text has been skipped.

Warnings

1. INTEGER CONSTANT TOO LARGE - 32767 USED

The priority value must fit in a half-word.

2. UNLABELLED STOP OR RETURN FOLLOWS A TRANSFER

The STOP or RETURN can never be executed.

3. UNLABELLED GOTO STATEMENT FOLLOWS A TRANSFER

The GOTO can never be executed.

4. ASSIGNMENT OPERATOR EXPECTED

In an assignment statement, the left-most operator should be := or = .

5. MODE HAS BEEN REDEFINED FOR THIS PROCEDURE

The user is redefining a mode name.

6. PRIORITY ALREADY DEFINED - NEW VALUE USED

The user is redefining an operator priority.

7. RESULTING MODE ALREADY DEFINED - NEW MODE USED

The user is redefining the resulting mode of an operator definition.

8. TEMPLATE ALREADY SUPPLIED - NEW ONE USED

The user is redefining the template in an operator definition.

```
//RUNFLEX JOB '0462,XXXX,MP,T=5,L=1,C=0',MARIAN                                    JOB 980
***MUMXXXX
***ROUTEPRINT NEMP
//FLEX EXEC PGM=FLEX,REGION=60K
//STEPLIB DD DSN=ZARNKE.M0295.FLEXLIB,DISP=SHR
//SYSPUNCH DD UNIT=SYSDA,SPACE=(TRK,(20,10)),DISP=(,PASS),
//             DCB=(RECFM=FB,BLKSIZE=4240,LRECL=80)
//SYSUT1 DD UNIT=SYSDA,SPACE=(TRK,(20,10))
//TRANSOUT DD SYSOUT=A,DCB=(RECFM=FA,BLKSIZE=133)
//TRANSIN DD *
IEF142I - STEP WAS EXECUTED - COND CODE 0012
IEF373I STEP /FLEX    / START 74242.2132
IEF374I STEP /FLEX    / STOP  74242.2132 CPU   0MIN 00.47SEC MAIN  58K LCS   OK
FLEX        20.54 SEC EXEC TIME      0.47 SEC CPU TIME      178 I/O COUNTS  60K REGION  58K USED

// EXEC FORTGCLG,COND=(8,LT)
//FORT.SYSPRINT DD DUMMY
//FORT.SYSIN DD DSN=*.FLEX.SYSPUNCH,DISP=(OLD,PASS)
IEF202I - STEP - FORT    , WAS NOT RUN BECAUSE OF CONDITION CODES.
IEF373I STEP /FORT    / START 74242.2132
IEF374I STEP /FORT    / STOP  74242.2132 CPU   0MIN 00.00SEC MAIN   0K LCS   OK
FORT         4.14 SEC EXEC TIME      0.00 SEC CPU TIME       41 I/O COUNTS 104K REGION   0K USED

IEF202I - STEP - LKED    , WAS NOT RUN BECAUSE OF CONDITION CODES.
IEF373I STEP /LKED    / START 74242.2132
IEF374I STEP /LKED    / STOP  74242.2133 CPU   0MIN 00.00SEC MAIN   0K LCS   OK
LKED         6.95 SEC EXEC TIME      0.00 SEC CPU TIME       80 I/O COUNTS 104K REGION   0K USED

IEF202I - STEP - GO      , WAS NOT RUN BECAUSE OF CONDITION CODES.
IEF373I STEP /GO      / START 74242.2133
IEF374I STEP /GO      / STOP  74242.2133 CPU   0MIN 00.00SEC MAIN   0K LCS   OK
GO           2.97 SEC EXEC TIME      0.00 SEC CPU TIME       39 I/O COUNTS  52K REGION   0K USED

IEF375I  JOB /RUNFLEX / START 74242.2132
IEF376I  JOB /RUNFLEX / STOP  74242.2133 CPU   0MIN 00.47SEC
RUNFLEX     34.60 SEC EXEC TIME      0.47 SEC CPU TIME        DATE 74.242 UNIVERSITY OF MANITOBA
```

```
     1
     2
     3
***STATEMENT      1,LINE     4***   ILLEGAL MODE NAME
***STATEMENT      1,LINE     4***   TEXT BETWEEN $'S HAS BEEN OMITTED
                $             $
     4     1  MODE 4EALA = REAL;
     5
     6
     7     2  OPERATOR ABC(REAL,INTEGER),
     8                 E1(REAL,REAL),
     9                 DEF(REAL,REAL)=REAL;
    10
    11
    12     3  PROGRAM;
    13     4     REAL X,Y,Z;
    14     5     INTEGER I,J,K;
    15     6     CALL ABC(X,I);
    16     7     Y:=DEF(X,Y);
    17     8     CALL E1(X,Y)
    18        ENDPROC;
    19
    20
    21     9  SUBROUTINE ABC(P1,P2);
    22    10     REAL P1,P3;
    23    11     INTEGER P2;
    24    12     P1:=2**P2-P3;
***STATEMENT     13,LINE    25***   PARAMETER NAME HAS BEEN USED PREVIOUSLY
                  $
    25    13  ENTRY E1(P3,Q);
    26    14     REAL Q            ,XYZ;
    27    15     XYZ:=  P1
    28        ENDPROC;
    29
    30
    31    16  FUNCTION REAL DEF(A,B);
***STATEMENT     17,LINE    32***   NAME PREVIOUSLY DEFINED - OLD DEFINITION USED
                  $
    32    17     REAL DEF,A,B;
    33    18     DEF:=A+B
    34        ENDPROC
```

APPENDIX 4

Two Sample Programs

This appendix contains listings and generated object code for two programs that illustrate many of the features of FLEX.

The first program defines a number of functions and subroutines which perform operations on polynomials of degree less than sixteen. The basic mode and operator definitions occupy the first page of the example. These definitions are global definitions; they would be used as a prelude to each program that the user writes to do polynomial manipulation. The next two pages define the procedures referred to in these global definitions. The fourth page contains a sample mainline program that applies the preceding definitions. The last four pages of the example contain the FORTRAN object code generated by FLEX.

The second example illustrates the solution of a linear Diophantine equation whose coefficients are multi-precise integers. The first two pages of the source program contain the global definitions that are used with the multiple precision package. They allow expressions involving multi-precise integers to be coded in the normal infix notation, thus making it very easy for a programmer to perform multiple precision operations. Comparison of the FLEX program with the FORTRAN program shows how confusing it would be to code the FORTRAN program directly using function calls.

Statement 5 illustrates how the = operator may be extended to compare logical values.

```
1  COMMENT*********************************************
   *
   *  GLOBAL DEFINITION AREA
   *
   *********************************************;

   %  DECLARE REAL POLYNOMIALS WITH A MAXIMUM DEGREE OF 15

2  MODE POLY:REAL(16) READ(POLYIN) WRITE(PLYOUT);

   %  DECLARE OPERATORS ON THE POLYNOMIALS

3  OPERATOR +      (POLY,POLY)=POLY
                   ('DO &S1 I$=1,16;'
                    '&S1&0(I$)=&1(I$)+&2(I$);'
                    '&0'),
            +      (POLY,REAL)=POLY
                   ('&0(1)=&1(1)+&2;'
                    'DO &S1 I$=2,16;'
                    '&S1&0(I$)=&1(I$);'
                    '&0'),
            -      (POLY)=POLY
                   ('DO &S1 I$=1,16;'
                    '&S1&0(I$)=-&1(I$);'
                    '&0'),
            *      ((REAL,INTEGER),POLY)=POLY
                   ('DO &S1 I$=1,16;'
                    '&S1&0(I$)=&1*&2(I$);'
                    '&0'),
            =      (POLY,POLY)=LOGICAL
                   ('EQUALP(&1,&2)'),
            =      (POLY,REAL)=LOGICAL
                   ('EQUAL(&1,&2)'),
            ¬=     (POLY,POLY)=LOGICAL
                   ('.NOT.EQUALP(&1,&2)'),
            ¬=     (POLY,REAL)=LOGICAL
                   ('.NOT.EQUAL(&1,&2)'),
            OF     (POLY,REAL)=REAL,
            DERIV  (POLY)=POLY
                   ('CALL DERIV(&1,&0);'
                    '&0'),
            :=     (POLY,POLY)=
                   ('DO &S1 I$=1,16;'
                    '&S1&1(I$)=&2(I$);'),
            :=     (POLY,REAL)=
                   ('&1(1)=&2;'
                    'DO &S1 I$=2,16;'
                    '&S1&1(I$)=0.;');

   %  THESE FUNCTIONS MUST BE DECLARED IN THE FLEX PROGRAM
   %  SO THAT THEY WILL BE DECLARED IN THE OBJECT PROGRAM

4  OPERATOR EQUAL (POLY,REAL)=LOGICAL,
            EQUALP (POLY,POLY)=LOGICAL;

   %  'OF' WILL BE USED AS AN INFIX OPERATOR

5  PRIORITY OF = 1000;
```

```
    6  COMMENT*****************************************
       *
       *  PROCEDURES FOR INPUT AND OUTPUT OF POLYNOMIALS
       *
       ***********************************************;

    7  SUBROUTINE POLYIN (F);

    8     COMMENT A POLYNOMIAL F OF DEGREE N REQUIRES N+2 VALUES FOR INPUT:
                       THE VALUE OF N,AND
                       THE N+1 COEFFICIENTS  IN INCREASING POWERS;

    9     POLY F;
   10     INTEGER I,N;
   11     READ, N,  % READ DEGREE OF POLYNOMIAL
             DO (I=1 TO N+1, F(I));  %READ THE COEFFICIENTS
   12     DO I=N+2 TO 16;
   13        F(I) = 0.  % ZERO REMAINING COEFFICIENTS
          END
       ENDPROC;


   14  SUBROUTINE PLYOUT (F);

   15     COMMENT THE INDEX I OF THE HIGHEST ORDER NON-ZERO COEFFICIENT
                  IS DETERMINED, AND I COEFFICIENTS ARE PRINTED.
                  IF F=0. F(1) IS PRINTED;

   16     POLY F;
   17     INTEGER I,J:
   18     DO I=16 DOWNTO 1
             WHILE F(I)=0.  % DETERMINE FIRST NON-ZERO COEFFICIENT
          END;
   19     WRITE, DO (J=I DOWNTO 2,
                     F(J), 'X**', J-1:I(2), '+'),
                 F(1);
       ENDPROC;


   20  COMMENT*****************************************
       *
       *  LOGICAL FUNCTIONS FOR POLYNOMIAL COMPARISON
       *
       ***********************************************;

   21  FUNCTION LOGICAL EQUALP (F,G);

   22     COMMENT TWO POLYNOMIALS F AND G ARE EQUAL IF F(I) = G(I)
                  FOR I=1 TO 16;

   23     POLY F,G;
   24     INTEGER I;
   25     DO I=1 TO 16
             WHILE F(I) = G(I)  % DETERMINE IF COEFFICIENTS ARE EQUAL
          END;
   26     IF I <= 16 THEN EQUALP = FALSE
   28                ELSE EQUALP = TRUE  FI
       ENDPROC;
```

```
29  FUNCTION LOGICAL EQUAL (F,A);

30     COMMENT A POLYNOMIAL F IS EQUAL TO A SCALAR A IF
               F IS OF DEGREE 0 AND F(1)=A;

31     POLY F;
32     REAL A;
33     INTEGER I;
34     IF F(1)  = A
35        THEN EQUAL = FALSE
36        ELSE DO I=2 TO 16 WHILE F(I) = 0. END;
37             IF I <= 16 THEN EQUAL = FALSE
39                        ELSE EQUAL = TRUE  FI
       FI
    ENDPROC;


40  COMMENT*******************************************
    *
    *  POLYNOMIAL EVALUATION
    *
    *********************************************;

41  FUNCTION REAL OF (F,X);

42     COMMENT EVALUATE POLYNOMIAL F AT THE POINT X;

43     POLY F;
44     REAL X;
45     INTEGER I;
46     OF = F(16);
47     DO I=15 DOWNTO 1;
48        OF = OF * X + F(I)
       END
    ENDPROC;


49  COMMENT*******************************************
    *
    *  FIRST DERIVATIVE
    *
    *********************************************;

50  SUBROUTINE DERIV (F,FPRIME);

51     COMMENT DERIVATIVE OF A POLYNOMIAL F IS A POLYNOMIAL FPRIME
               WITH DEGREE 1 LESS THAN DEGREE OF F;

52     POLY F,FPRIME;
53     INTEGER I;
54     FPRIME(16) = 0.;
55     DO I=15 DOWNTO 1;
56        FPRIME(I) = I * F(I+1)
       END
    ENDPROC;
```

```
 57   COMMENT**********************************************
      *
      *  MAINLINE PROCEDURE:
      *  THREE EXAMPLES USING THE DEFINITIONS GIVEN ABOVE
      *
      **********************************************;

 58   PROGRAM;

 59   COMMENT*********  EXAMPLE 1  ********************;

 60      COMMENT FOR POLYNOMIALS A AND B CHECK IF
                 DERIV (A) + DERIV (B) = DERIV (A + B);
 61      POLY A,B,C,D;
 62      READ (5) A,B;
 63      C = DERIV A + DERIV B;
 64      D = DERIV (A + B);
 65      IF C ¬= D
 66         THEN WRITE (6) 'PLUS FORMULA NOT VERIFIED'
 67         ELSE WRITE (6) 'PLUS FORMULA VERIFIED'
         FI;
 68      WRITE (6) "A' + B' = ", C , :SKIP(1),
                "(A + B)' = ", D ;

 69   COMMENT*********  EXAMPLE 2  ********************;

 70      COMMENT FOR POLYNOMIAL A AND SCALAR S CHECK IF
                 S DERIV (A) = DERIV (S A);
 71      REAL S;
 72      READ (5) S;
 73      C = S * DERIV A ;
 74      D = DERIV (S * A);
 75      IF C ¬= D
 76         THEN WRITE (6) 'SCALAR MULTIPLICATION FORMULA NOT VERIFIED'
 77         ELSE WRITE (6) 'SCALAR MULTIPLICATION FORMULA VERIFIED'
         FI;
 78      WRITE (6) "S * A' = ", C , :SKIP(1),
                "(S * A)' = ", D ;

 79   COMMENT*********  EXAMPLE 3  ********************;

 80      COMMENT TABULATE VALUES OF F(X) AND FPRIME(X) IN THE INTERVAL
                 -10 TO 10  TO GET AN IDEA OF THE SHAPE OF F, USING
                 FIRST DERIVATIVE TEST - CRITICAL POINTS LIE IN
                 INTERVALS WHERE SIGN CHANGES;
 81      POLY F;
 82      REAL X;
 83      READ (5) F;
 84      WRITE (6) :COL(10), 'F(X)', :COL(30), 'F''(X)',
                DO ( X = -10 TO 10 BY .5,
                   :SKIP,
                   :COL(5), F OF X : E(14,7),
                   :COL(24), DERIV F OF X : E(14,7)  );

      ENDPROC
      SUBROUTINEPOLYIN(/F/)
      LOGICAL*4EQUALP
      INTEGER*4T$00A,I,N
      LOGICAL*4EQUAL
      REAL*4F(16)
      REAL*40F
      CALL IO$ST(82,5)
      CALLIO$I4 (N)
      I=(1)
      T$00A=(N+(1))
      GOTO3
    4 I=(I+1)
    3 IF(I.GT.T$00A)GOTO5
      CALLIO$R4 (F(I))
      GOTO4
    5 CALL IO$END
      I=(N+(2))
      T$00A=(16)
      GOTO6
    7 I=(I+1)
    6 IF(I.GT.T$00A)GOTO8
      F(I)=(0.E0)
      GOTO7
    8 RETURN
      END
      SUBROUTINEPLYOUT(/F/)
      LOGICAL*4EQUALP
      INTEGER*4T$00A,J,I
      LOGICAL*4EQUAL
      REAL*4F(16)
      REAL*40F
      I=(16)
      T$00A=(1)
      GOTO3
    4 I=(I-1)
    3 IF(I.LT.T$00A)GOTO5
      IF(.NOT.(F(I).EQ.(0.E0)))GOTO5
      GOTO4
    5 CALL IO$ST(146,6)
      J=I
      T$00A=(2)
      GOTO6
    7 J=(J-1)
    6 IF(J.LT.T$00A)GOTO8
      CALLIO$R4 (F(J))
      CALLIO$CH (3, 'X**' )
      CALLIO$I4 ((J-(1)),(2))
      CALLIO$CH (1, '+' )
      GOTO7
    8 CALLIO$R4 (F((1)))
      CALL IO$END
      RETURN
      END
      LOGICALFUNCTIONEQUALP*4(/F/,/G/)
      INTEGER*4T$00A,I
      LOGICAL*4EQUAL
```

```
   REAL*4F(16),G(16)
   REAL*4OF
   I=(1)
   T$00A=(16)
   GOTO3
 4 I=(I+1)
 3 IF(I.GT.T$00A)GOTO5
   IF(.NOT.(F(I).EQ.G(I)))GOTO5
   GOTO4
 5 IF(.NOT.(I.LE.(16)))GOTO6
   EQUALP= .FALSE.
   GOTO7
 6 EQUALP= .TRUE .
 7 RETURN
   END
   LOGICALFUNCTIONEQUAL*4(/F/,/A/)
   LOGICAL*4EQUALP
   INTEGER*4T$00B,I
   REAL*4F(16)
   REAL*4A
   REAL*4OF
   IF(.NOT.(F((1)).EQ.A))GOTO3
   EQUAL= .FALSE.
   GOTO4
 3 I=(2)
   T$00B=(16)
   GOTO5
 6 I=(I+1)
 5 IF(I.GT.T$00B)GOTO7
   IF(.NOT.(F(I).EQ.(0.E0)))GOTO7
   GOTO6
 7 IF(.NOT.(I.LE.(16)))GOTO8
   EQUAL= .FALSE.
   GOTO9
 8 EQUAL= .TRUE .
 9 CONTINUE
 4 RETURN
   END
   REALFUNCTIONOF*4(/F/,/X/)
   LOGICAL*4EQUALP
   INTEGER*4T$00A,I
   LOGICAL*4EQUAL
   REAL*4F(16)
   REAL*4X
   OF=F((16))
   I=(15)
   T$00A=(1)
   GOTO3
 4 I=(I-1)
 3 IF(I.LT.T$00A)GOTO5
   OF=((OF*X)+F(I))
   GOTO4
 5 RETURN
   END
   SUBROUTINEDERIV(/F/,/FPRIME/)
   LOGICAL*4EQUALP
```

```
      INTEGER*4T$00A,I
      LOGICAL*4EQUAL
      REAL*4FPRIME(16),F(16)
      REAL*4OF
      FPRIME((16))=(0.E0)
      I=(15)
      T$00A=(1)
      GOTO3
    4 I=(I-1)
    3 IF(I.LT.T$00A)GOTO5
      FPRIME(I)=(I*F((I+(1))))
      GOTO4
    5 RETURN
      END
      LOGICAL*4EQUALP
      LOGICAL*4EQUAL
      REAL*4T$00A(16),T$00B(16),T$00C(16),D(16),A(16),F(16),B(16),C(16)
      REAL*4T$00E,T$00F,S,X
      REAL*4OF
      CALL IO$ST(80,(5))
      CALLPOLYIN(A)
      CALLPOLYIN(B)
      CALL IO$END
      CALL DERIV(A,T$00A)
      CALL DERIV(B,T$00B)
      DO 4 I$=1,16
    4 T$00C (I$)=T$00A (I$)+T$00B (I$)
      DO 14 I$=1,16
   14 C (I$)=T$00C (I$)
      DO 24 I$=1,16
   24 T$00C (I$)=A (I$)+B (I$)
      CALL DERIV(T$00C,T$00B)
      DO 34 I$=1,16
   34 D (I$)=T$00B (I$)
      IF(.NOT.(.NOT.EQUALP(C,D)))GOTO43
      CALL IO$ST(144,(6))
      CALLIO$CH (25, 'PLUS FORMULA NOT VERIFIED' )
      CALL IO$END
      GOTO44
   43 CALL IO$ST(144,(6))
      CALLIO$CH (21, 'PLUS FORMULA VERIFIED' )
      CALL IO$END
   44 CALL IO$ST(144,(6))
      CALLIO$CH (10, 'A'' + B'' = ' )
      CALLPLYOUT(C)
      CALLIO$SK ((1))
      CALLIO$CH (11, '(A + B)'' = ' )
      CALLPLYOUT(D)
      CALL IO$END
      CALL IO$ST(80,(5))
      CALLIO$R4 (S)
      CALL IO$END
      CALL DERIV(A,T$00C)
      DO 46 I$=1,16
   46 T$00B (I$)=S*T$00C (I$)
      DO 56 I$=1,16
```

```
 56 C (I$)=T$00B (I$)
    DO 66 I$=1,16
 66 T$00C (I$)=S*A (I$)
    CALL DERIV(T$00C,T$00B)
    DO 76 I$=1,16
 76 D (I$)=T$00B (I$)
    IF(.NOT.(.NOT.EQUALP(C,D)))GOTO85
    CALL IO$ST(144,(6))
    CALLIO$CH (42, 'SCALAR MULTIPLICATION FORMULA NOT VERIFIED' )
    CALL IO$END
    GOTO86
 85 CALL IO$ST(144,(6))
    CALLIO$CH (38, 'SCALAR MULTIPLICATION FORMULA VERIFIED' )
    CALL IO$END
 86 CALL IO$ST(144,(6))
    CALLIO$CH (9, 'S * A'' = ' )
    CALLPLYOUT(C)
    CALLIO$SK ((1))
    CALLIO$CH (11, '(S * A)'' = ' )
    CALLPLYOUT(D)
    CALL IO$END
    CALL IO$ST(80,(5))
    CALLPOLYIN(F)
    CALL IO$END
    CALL IO$ST(144,(6))
    CALLIO$COL((10))
    CALLIO$CH (4, 'F(X)' )
    CALLIO$COL((30))
    CALLIO$CH (5, 'F''(X)' )
    X=(-10)
    T$00E=(10)
    T$00F=(.5E0)
    GOTO87
 88 X=(X+T$00F)
 87 IF(X.GT.T$00E)GOTO89
    CALLIO$SK
    CALLIO$COL((5))
    CALLIO$R4 ((OF(F,X)),(14),(7))
    CALLIO$COL((24))
    CALL DERIV(F,T$00C)
    CALLIO$R4 ((OF(T$00C,X)),(14),(7))
    GOTO88
 89 CALL IO$END
    STOP
    END
 1  COMMENT***************************************
    *
    *  DEFINITION OF MULTIPLE PRECISION INTEGERS
    *
    ***************************************;


 2  MODE INTEGER*M : INTEGER*4
         READ(IO$IM) WRITE(IO$IM) INPUT(IO$IM) OUTPUT(IO$IM);


 3  OPERATOR +  (INTEGER*M,INTEGER*M)=INTEGER*M
                ('MPA(&1,&2)'),
            +   (INTEGER*M,INTEGER*4)=INTEGER*M
                ('MPAW(&1,&2)'),
            +   (INTEGER*4,INTEGER*M)=INTEGER*M
                ('MPAW(&2,&1)'),


            -   (INTEGER*M,INTEGER*M)=INTEGER*M
                ('MPS(&1,&2)'),
            -   (INTEGER*M,INTEGER*4)=INTEGER*M
                ('MPSW(&1,&2)'),
            -   (INTEGER*4,INTEGER*M)=INTEGER*M
                ('MPN(MPSW(&2,&1))'),
            -   (INTEGER*M)=INTEGER*M
                ('MPN(&1)'),


            *   (INTEGER*M,INTEGER*M)=INTEGER*M
                ('MPM(&1,&2)'),
            *   (INTEGER*M,INTEGER*4)=INTEGER*M
                ('MPMW(&1,&2)'),
            *   (INTEGER*4,INTEGER*M)=INTEGER*M
                ('MPMW(&2,&1)'),


            /   (INTEGER*M,INTEGER*M)=INTEGER*M
                ('MPD(&1,&2)'),
            /   (INTEGER*M,INTEGER*4)=INTEGER*M
                ('MPDW(&1,&2)'),


            //  (INTEGER*M,INTEGER*M)=INTEGER*M
                ('MPR(&1,&2)'),
            //  (INTEGER*M,INTEGER*4)=INTEGER*4
                ('MPRW(&1,&2)'),


            **  (INTEGER*M,INTEGER*4)=INTEGER*M
                ('MPXW(&1,&2)'),


            MP  (INTEGER*4)=INTEGER*M
                ('MPW(&1)'),
=   (INTEGER*M,INTEGER*M)=LOGICAL*4
    ('MPCEQ(&1,&2)'),
=   (INTEGER*M,INTEGER*4)=LOGICAL*4
    ('MPCEQW(&1,&2)'),
=   (INTEGER*4,INTEGER*M)=LOGICAL*4
    ('MPCEQW(&2,&1)'),


¬=  (INTEGER*M,INTEGER*M)=LOGICAL*4
    ('MPCNE(&1,&2)'),
¬=  (INTEGER*M,INTEGER*4)=LOGICAL*4
    ('MPCNEW(&1,&2)'),
¬=  (INTEGER*4,INTEGER*M)=LOGICAL*4
    ('MPCNEW(&2,&1)'),


>   (INTEGER*M,INTEGER*M)=LOGICAL*4
    ('MPCGT(&1,&2)'),
>   (INTEGER*M,INTEGER*4)=LOGICAL*4
    ('MPCGTW(&1,&2)'),
>   (INTEGER*4,INTEGER*M)=LOGICAL*4
    ('MPCLTW(&2,&1)'),


>=  (INTEGER*M,INTEGER*M)=LOGICAL*4
    ('MPCGE(&1,&2)'),
>=  (INTEGER*M,INTEGER*4)=LOGICAL*4
    ('MPCGEW(&1,&2)'),
>=  (INTEGER*4,INTEGER*M)=LOGICAL*4
    ('MPCLEW(&2,&1)'),


<   (INTEGER*M,INTEGER*M)=LOGICAL*4
    ('MPCLT(&1,&2)'),
<   (INTEGER*M,INTEGER*4)=LOGICAL*4
    ('MPCLTW(&1,&2)'),
<   (INTEGER*4,INTEGER*M)=LOGICAL*4
    ('MPCGTW(&2,&1)'),


<=  (INTEGER*M,INTEGER*M)=LOGICAL*4
    ('MPCLE(&1,&2)'),
<=  (INTEGER*M,INTEGER*4)=LOGICAL*4
    ('MPCLEW(&1,&2)'),
<=  (INTEGER*4,INTEGER*M)=LOGICAL*4
    ('MPCGEW(&2,&1)'),


:=  (INTEGER*M,INTEGER*M)=
    ('CALL MPE(&1,&2);'),
:=  (INTEGER*M,INTEGER*4)=
    ('CALL MPEW(&1,&2);'),
:=  (INTEGER*4,INTEGER*M)=
    ('&1=MPI(&2);');
```

```



 4   COMMENT*********************************************************
     *
     *  THIS PROGRAM SOLVES THE LINEAR DIOPHANTINE EQUATION
     *                  A X + B Y = C
     *  WHERE A, B, C ARE MULTI-PRECISE.
     *. IT USES THE STANDARD TECHNIQUE OF EXPRESSING
     *                  A / B
     *  AS A CONTINUED FRACTION TO FIND THE SOLUTION OF
     *                  A X + B Y = (A,B).
     *
     ********************************************************;



 5   OPERATOR = (LOGICAL*4,LOGICAL*4)=LOGICAL*4
                ('(&1.AND.&2).OR..NOT.(&1.OR.&2)');


 6   PROGRAM;


 7       INTEGER*M A, X, B, Y, C,
                   APREV, BPREV,
                   XPREV, XNEXT, YPREV, YNEXT,
                   Q, R,
                   MULT;
 8       INTEGER*4 SIGN, JSIGN;


 9       SIGN := +1;

10       READ (5) A, B, C;
11       APREV := A;
12       BPREV := B;


13       IF B = 0
14          THEN IF A = 0
15                 THEN WRITE (6) 'ILLEGAL DIOPHANTINE EQUATION';
16                      STOP
                   FI;
17               B := A;
18               X := 1;
19               Y := 0;


20          ELSE COMMENT SET X(1), X(0), Y(1), Y(0) = 0, 1, 1, 0
                         IF SIGNS OF A AND B ARE THE SAME
                                               OR = 0, -1, -1, 0
                         IF SIGNS OF A AND B ARE DIFFERENT *******;

21               IF (A > 0) = (B > 0) THEN JSIGN := 1
23                                    ELSE JSIGN := -1  FI;
24               X := 0;
25               XPREV := JSIGN;
26               Y := JSIGN;
27               YPREV := 0;
```

```
28            COMMENT FIND  X(N+1) = Q X(N) + X(N-1)
                            Y(N+1) = Q Y(N) + Y(N-1)
                     CONTINUE UNTIL REMAINDER IS ZERO ********;

29            R := A // B;
30            WHILE R ¬= 0;
31                  Q := A / B;
32                  XNEXT := X * Q + XPREV;
33                  XPREV := X;
34                  X     := XNEXT;
35                  YNEXT := Y * Q + YPREV;
36                  YPREV := Y;
37                  Y     := YNEXT;
38                  A := B;
39                  B := R;
40                  SIGN := -SIGN;
41                  R := A // B
              END
      FI;


42    IF C // B ¬= 0
43       THEN WRITE (6) 'NO SOLUTION FOR DIOPHANTINE EQUATION';
44            STOP
      FI;


45    MULT := (C / B);
46    X := X * MULT;
47    Y := Y * MULT;
48    IF SIGN = +1 THEN X = -X
50                ELSE Y = -Y  FI;
51    IF APREV * X + BPREV * Y ¬= C
52       THEN WRITE (6) 'SOLUTION IS INCORRECT'
53       ELSE WRITE (6) APREV, BPREV, C, X, Y
      FI
   ENDPROC
```

```
      INTEGER*4BPREV,Y,MULT,XPREV,YPREV,A,Q,XNEXT,B,R,APREV,YNEXT,X,C
      INTEGER*4JSIGN,SIGN
      SIGN=(+1)
      CALL IO$ST(80,(5))
      CALLIO$IM (A)
      CALLIO$IM (B)
      CALLIO$IM (C)
      CALL IO$END
      CALL MPE(APREV,A)
      CALL MPE(BPREV,B)
      IF(.NOT.(MPCEQW(B,(0))))GOTO3
      IF(.NOT.(MPCEQW(A,(0))))GOTO5
      CALL IO$ST(144,(6))
      CALLIO$CH (28, 'ILLEGAL DIOPHANTINE EQUATION' )
      CALL IO$END
      STOP
    5 CALL MPE(B,A)
      CALL MPEW(X,(1))
      CALL MPEW(Y,(0))
      GOTO4
    3 IF(.NOT.(((MPCGTW(A,(0))).AND.(MPCGTW(B,(0)))).OR..NOT.((MPCGTW(A,
     X(0))).OR.(MPCGTW(B,(0))))))GOTO7
      JSIGN=(1)
      GOTO8
    7 JSIGN=(-1)
    8 CALL MPEW(X,(0))
      CALL MPEW(XPREV,JSIGN)
      CALL MPEW(Y,JSIGN)
      CALL MPEW(YPREV,(0))
      CALL MPE(R,(MPR(A,B)))
    9 IF(.NOT.(MPCNEW(R,(0))))GOTO10
      CALL MPE(Q,(MPD(A,B)))
      CALL MPE(XNEXT,(MPA((MPM(X,Q)),XPREV)))
      CALL MPE(XPREV,X)
      CALL MPE(X,XNEXT)
      CALL MPE(YNEXT,(MPA((MPM(Y,Q)),YPREV)))
      CALL MPE(YPREV,Y)
      CALL MPE(Y,YNEXT)
      CALL MPE(A,B)
      CALL MPE(B,R)
      SIGN=(-SIGN)
      CALL MPE(R,(MPR(A,B)))
      GOTO9
   10 CONTINUE
    4 IF(.NOT.(MPCNEW((MPR(C,B)),(0))))GOTO11
      CALL IO$ST(144,(6))
      CALLIO$CH (36, 'NO SOLUTION FOR DIOPHANTINE EQUATION' )
      CALL IO$END
      STOP
   11 CALL MPE(MULT,(MPD(C,B)))
      CALL MPE(X,(MPM(X,MULT)))
      CALL MPE(Y,(MPM(Y,MULT)))
      IF(.NOT.(SIGN.EQ.(+1)))GOTO13
      CALL MPE(X,(MPN(X)))
      GOTO14
   13 CALL MPE(Y,(MPN(Y)))
```

```
14 IF(.NOT.(MPCNE((MPA((MPM(APREV,X)),(MPM(BPREV,Y)))),C)))GOTO15
   CALL IO$ST(144,(6))
   CALLIO$CH (21, 'SOLUTION IS INCORRECT' )
   CALL IO$END
   GOTO16
15 CALL IO$ST(144,(6))
   CALLIO$IM (APREV)
   CALLIO$IM (BPREV)
   CALLIO$IM (C)
   CALLIO$IM (X)
   CALLIO$IM (Y)
   CALL IO$END
16 STOP
   END
```

REFERENCES

[1] Backus, J.W. *The syntax and semantics of the proposed international algebraic language of the Zürich ACM-GAMM conference.* Proc. ICIP, UNESCO (June 1959) 125-132.

[2] IBM. *System /360 and System /370 FORTRAN IV Language.* Form C28-6515.

[3] IBM. *Operating System Assembler Language.* Form C28-6514.

[4] Lindsey, C.H., and van der Meulen, S.G. *Informal Introduction to ALGOL 68.* North-Holland Publishing Company, Amsterdam (1971).

[5] Mills, H.D. *Mathematical foundations for structured programming.* Report FSC72-6012, IBM Federal Systems Division, Gaithersburg, Md. (February 1972).

[6] Zarnke, C.R. *Routines for multi-precise integer arithmetic.* Scientific Report No. 69, Department of Computer Science, The University of Manitoba (April 1973).

[7] Zarnke, C.R., and Zajac, B.P. *Input, output and debugging routines for FLEX.* Scientific Report, Department of Computer Science, The University of Manitoba (to appear).

[8] Clark, B.L. *The project SUE system language users guide.* Computer Systems Research Group, University of Toronto (September 1973).

[9] Wirth, N. *PL/360, a programming language for the 360 computer.* JACM, Vol. 15 (1967) 37-74.