MYTRAN:

A PROGRAMMING LANGUAGE FOR DATA ABSTRACTION

MYTRAN:

A PROGRAMMING LANGUAGE FOR DATA ABSTRACTION

By

TIMOTHY WEST SNIDER, B.Sc.

A Project

Submitted to the School of Graduate Studies

in Partial Fulfilment of the Requirements

for the Degree

Master of Science

McMaster University

1981

MASTER OF SCIENCE (Computer Science)

MCMASTER UNIVERSITY
Hamilton, Ontario

TITLE: Mytran: A programming Language for Data Abstraction

AUTHOR: Timothy West Snider, B.Sc. (McMaster University)

SUPERVISOR: Professor Derick Wood

NUMBER OF PAGES: v, 84, A-12, B-9, C-9, D-67.

Abstract

This project is about the design and implementation of a new programming language, Mytran. Two new control statements are implemented in Mytran. Data abstraction is supported through parameterized types or "type constructors".

CONTENTS

Chapter I

Introduction

This project is about the design and implementation of a language which supports data abstraction. The design is based on intuitive ideas of what a programming language should do. The implementation is as a Fortran preprocessor. In this chapter, I will outline the goals and objectives of the project by answering two questions. First, why design a language to support data abstraction. Second, why implement it as a Fortran preprocessor.

To answer the first question, we need first of all to define what is meant by the term data abstraction. Hoare [1972] defines abstraction as the "recognition of similarities between certain objects ... and the decision to concentrate on these similarities and to ignore ... the differences".

Thus a set of objects may be represented by a general definition or abstraction, which only describes the similar aspects of the objects and leaves the unique aspects of each object open to later description. For example, a book, to a student, is a collection of pages,

bound in a cover. The nature of the information in the book, whether it is words or maps or tables, or whether it is blank and is to have words or maps or tables put into it, is not included in the abstraction. The specific cases of text book, atlas, book of tables, or note book, all add additional information to the book abstraction. However, the operations of opening the book to the first or last page, turning a page forwards or backwards, closing the book, etc., are all included in the abstract definition of a book, and do not change in the specific cases.

From different points of view, the same set of objects may have a different abstraction. To a bookstore, the student's books are items that take up shelf space, sell at certain prices, must be reordered when they sell out, etc.

In programming, abstraction allows us to define complex objects or models in independent layers. Continuing with our abstract books, the operations to be performed on a book, may be programmed without regard to the many forms of information that it may hold. And the operations on a page may be programmed without regard to the fact that it will be held in a book. In fact, the

less they know about each other, the better. A certain minimum interface is necessary. For example, let us say that one of the operations to be performed on a book, is a search for a word. The book abstraction defines the opening and turning of pages. But it must pass the word to the page abstraction to see if it is contained in a page. Thus if this search operation is to be part of the abstract definition of a book, each different abstract definition of a page must include the ability to search its contents.

Abstraction has been supported by programming languages, via the subprogram, right from the beginning. A set of statements may be collected together under one name, and their execution invoked by referencing that name. Subprograms may be written to perform general operations on classes of objects through the use of parameters. Thus, by a careful discipline of programming, it is possible to define independent data abstractions in any language. Obviously, as the number of language facilities supporting data abstraction decreases, the difficulty of maintaining the discipline increases. For example, Fortran has only one language mechanism for handling collections of data, namely the array. It is a true abstraction in that an array may contain any type of

scalar information, integer, real, logical etc., for which the access remains the same and individual components may be operated on without reference to the fact that they are in an array. However, it requires an extremely messy discipline to implement anything approaching Pascal-type records in Fortran. Wirth [1980] gives a history of the development of data abstraction in programming languages. In particular, he comments that few good implementations exist. Those that do exist are not widely available. This makes the design and implementation of a new language a worthwhile goal.

The second question is, why implement a new language via a Fortran preprocessor. This may be considered in two parts, first why implement a new language via a preprocessor, and second why use Fortran as the target language. The reasons for using a preprocessor are related to the fact that this language design is experimental. This means that many attributes of a programming language are not of concern to us. For example the method of passing parameters, or the method of storage allocation are not of interest. A preprocessor allows new language features to be added to the source language, while allowing any of the required attributes of the target language to be retained. Also, the efficient

generation of efficient code is not a requirement. So a preprocessor is a relatively easy technique for implementing a new language.

The reason for using Fortran as the target language is precisely the same reason that it is currently considered unacceptable by many programmers. It has no rigorous type-checking of parameters, and allows very easy use of lots of statement labels. These two properties which make reading and checking of programs so confusing, are absolutely necessary for generating code. Another advantage of Fortran is that it is widely available. This combined with the preprocessor approach, means that a language implemented in this manner is easily made available on a wide range of computers.

So the goal of this project is to produce a language that supports data abstraction, and is reasonably portable. We will also design some new control statements since those in Fortran do not readily support structured programming. Since this is an experimental language, we will try to design new control statements rather than implement some that already exist in other languages.

The rest of this report describes the design and

implementation processes. Chapter 2 discusses preprocessor techniques and examples. Chapter 3 discusses principles and examples of control statements. Chapter 4 does the same for data structures. Chapter 5 discusses the implementation techniques used. Chapter 6 discusses the successes and failures of the project.

Chapter II

Preprocessor Techniques

A preprocessor is a translator. It converts a program written in a source language into one written in a target language. A compiler is also a translator and though there are some translators which are clearly considered to be compilers and others that are clearly considered to be preprocessors, there is no precise definition that separates the two. On the one hand a compiler normally translates from a high level language to an executable or near executable language and there is a large difference between its source and target languages. On the other hand a preprocessor normally translates from a high level language to another high level language. There are usually only small differences between its source and target languages but this is not always the case. For example, a preprocessor might be used to translate a program written in an extended version of Pascal into a program written in standard Pascal. The Pascal compiler would then be used to translate the program into executable code. However a preprocessor could also be used to translate from Algol to Fortran, which are very different source and target languages.

Solntseff and Yezerski [1974] give a very detailed analysis of the different stages of translation and the nature of the extensions that can be made at each of these stages. For the remainder of this chapter, preprocessors are considered to be simple translators, falling into Solntseff and Yezerski's lexical and syntactic extension classes.

Preprocessors of this type are used to add new features or otherwise enhance an existing language. This is much cheaper than writing or modifying a compiler. The preprocessor need only examine the source program for those features that it is adding. Any features already in the language may just be copied to the target program, to be handled by the compiler.

We now look at some general classes of preprocessors and their properties as well as some examples. This is not intended to be an exhaustive description of preprocessors but rather it is a brief introduction to them.

2.1 Macro Preprocessors

A macro preprocessor translates from source to target language by using patterns of text. There is a source

pattern, and a target pattern. The patterns consist of fixed text and may or may not include text collecting and generating parameters.

The fixed text in the source pattern may take the form of keywords which set the boundaries of the parameters, as in

(1) ADD %1 TO %2 GIVING %3

Or the fixed text may just be a macro name, with parameters separated by commas, as in

(2) SUM %1, %2, %3

In the source pattern, the parameters collect text.

A macro is invoked by using its name, or its first keyword. The pattern is then matched and source text is assigned to the parameters as they occur in the pattern. Thus, pattern (1) would be invoked by

(3) ADD A TO B GIVING C

and pattern (2) would be invoked by

(4) SUM A, B, C

In either case, parameters %1, %2, and %3 would be assigned the text A, B, and C respectively.

In the target pattern, the fixed text consists of full or partial statements in the target language. The parameters, which now generate text, are interspersed within the fixed text, to complete partial statements or generate entirely new statements. A reasonable target pattern to use with source patterns (1) or (3) is

```
(5)  LOAD %1
     ADD  %2
     STORE %3
```

An equally valid target pattern is

```
(6)  %3 = %1 + %2
```

The target code is generated by copying the fixed text and when a parameter is encountered, the text collected by the source pattern is generated. Thus, pattern (5), whether used with source pattern (1) or (3), when invoked by the appropriate statement will generate

```
(7)  LOAD  A
     ADD   B
     STORE C
```

Also, pattern (6), used with either source pattern will generate

```
(8)  C = A + B
```

Macro preprocessors were originally used with assembly languages and are still most commonly associated

with them. However, as we have seen, a general purpose macro processor makes no requirements on its source and target languages. In fact, it may be used on any file of text.

A number of Fortran macro processors are available. One that was written for use with Fortran, but is in fact general purpose, is Mortran2 [Cook 1975]. It is normally used with a preamble of macros which implement a structured version of Fortran. The programmer may then add to or override this set. A typical macro in the Mortran2 preamble is

```
(9)  %'UNTIL # < # >' =
       ':1: IF #1 GOTO :2:; #2; GOTO :1:; :2: CONTINUE'
```

Mortran2 uses colons to indicate labels and semi-colons to separate statements. This macro will cause the source statement

```
(10)  UNTIL (I .GT. 20) <I=I+1; PRINT i*i>
```

to be translated into

```
(11)  1 IF (I .GT. 20) GOTO 2
        I=I+1
        PRINT I*I
        GOTO 1
      2 CONTINUE
```

Lexical macro preprocessors have been used for a long time. They have become a powerful language extension

tool. One problem with them is that since they know nothing of the syntax of their source languages, they cannot even recognize errors, let alone recover from them. Another problem is that the syntax for defining macros is usually very different from the syntax of the source language. This adds to the complexity of the programming task and adds confusion to the reading of programs. Much more complete discussions of macros are given in [Solntseff 1974], [Brown 1974], [Campbell-Kelly 1973].

2.2 Statement Preprocessors

This class of preprocessors adds new statement types to an existing language. When the term preprocessor is used, it most often refers to this type of system. In fact the most common use of the preprocessor concept is to add new statement types to Fortran to allow structured programming techniques.

These preprocessors work like restricted macro processors. In the terminology of Solntseff and Yezerski they are called syntactic extension mechanisms. Source patterns and target patterns are used but they are built into the system. For example, the IF statement in DEFT [Steele 1974] uses the source pattern

```
(12)  IF %1 THEN %2 ELSE %3 END
```

and the target pattern

```
(13)      IF %1 GOTO L1
          GOTO L2
      L1  %2
          GOTO L3
      L2  %3
      L3  CONTINUE
```

DEFT checks the syntax of this statement and its parameters. For instance, if parentheses are not balanced in a condition, this will be flagged as an error, and some recovery attempted.

A wide variety of statements is supported by this class of system, usually one or more variations of the IF-THEN-ELSE statement and several loop structures. IFTRAN allows the following statement

```
(14)  WHILE (I .LT. J)
      I = I + 1
      END WHILE
```

which generates

```
(15)  99999 CONTINUE
            IF (.NOT.(I .LT. J)) GOTO 99998
            I = I + 1
            GOTO 99999
      99998 CONTINUE
```

Ratfor [Kernighan] allows

```
(16)   FOR (I=1; I<2Ø; I=I+1)
       PRINT I*I
```

which generates

```
(17)         I=1
             GOTO 99999
       99998 CONTINUE
             I=I+1
       99999 IF (.NOT.(I.LT.2Ø)) GOTO 99997
             PRINT I*I
             GOTO 99998
       99997 CONTINUE
```

As with these examples, the translation of individual statements is always very simple and straightforward. However, when structures are nested more than two or three levels deep, it becomes quite difficult to duplicate the translation manually.

Many of these systems also support other features, for example long variable names, alphanumeric labels, or free format statement entry.

2.3 Miscellaneous Preprocessors

There are as many of these as there are ideas about what a programming language should do.

Many algorithms are most naturally expressed recursively. It is quite difficult to write some of these

in a language which does not support recursion. Another large class of algorithms uses the concept of coroutines. Both of these concepts involve a different way of handling the linkage between subprograms. There are several preprocessors which support these capabilities. They use new statements to indicate when the alternate linkages are to be used. For example, the Star [Arisawa 1979] system uses the statements RECCAL and RECRET to indicate a recursive call and a recursive return. These new statements are translated in a straightforward manner into the target language. This is much the same as the statement preprocessors. However, these statements also generate calls to system routines to maintain return address and variable stacks. The system described by Skordilakis [1978] uses the RESUME statement to restart a coroutine at the point at which it was last interrupted. The statement

```
(18)  RESUME A
```

is translated into

```
(19)       IP = 2
           CALL SYS (2,2,3)
           CALL A
      902 CONTINUE
```

IP is a local variable which is used by a computed GOTO to branch to statement 902, when this routine is RESUMEd.

SYS keeps track of return addresses. The parameters are codes for the type of statement, the calling routine, and the called routine. Although the translation is quite simple, the net effect is quite complex.

Another common problem is that of handling non-standard data types. This involves defining storage for variables, and also new operators to use with them. The Augment system [Crary 1974] allows the programmer to define subprograms to perform operations on a non-standard data type. These subprograms are then associated with one of the standard infix operators, or possibly with some new operator symbol. For example, a routine to multiply two double precision complex numbers may be written, and associated with the symbol *. When a statement such as

```
(2Ø)  A = B * C
```

is encountered in the source, where A, B, and C have been declared to be double precision complex, a call to the subroutine is generated. If the multiplication occurs as part of a longer expression, the call will be generated using a temporary location for the result. This temporary location will then be passed as an operand to some other routine. This means that support packages for non-standard data types may be written. Then algorithms

written for standard data types, may be recompiled in conjunction with the appropriate support package. If all the necessary routines are in the package, the algorithm will then work on the new data type.

2.4 Summary

We have seen that a wide variety of programming language features may be implemented by preprocessing. The major disadvantage of all of these systems is that they are an extra layer between the programmer and the computer. That means an extra layer of processing to get a source program into executable form and an extra layer of tracing during debugging. The latter can be quite frustrating, since many systems have only recently begun to give high level language tracebacks and snapshots of error points, which unfortunately refer to the target language of the preprocessor, not the source. The advantage is that preprocessing is a fast and cheap way to provide new language features.

Chapter III

Control Statements

Control statements alter the sequence of execution of other statements. The statement

```
IF (condition)
   block1
ELSE
   block2
ENDIF
```

alters the sequence of execution in one of two possible ways. Either the first block is executed and the second block is skipped. Or the first block is skipped, and the second executed. There are two types of control statements used in structured programming. One type controls the selection of statements for execution. The other controls repeated execution of statements. The IF-ELSE-ENDIF statement above is an example of a selective control statement.

Control statements do not change the values of any program variables. They determine which statements that do change program variables will be executed. So they have a much more far reaching effect on the results of a program than other executable statements. Because of

this, it is quite important that the effect of a control statement be very clear.

3.1 Selective Control Statements

A selective control statement determines which of the statements under its control will be executed. The simplest form of this is the Fortran IF statement.

```
IF (condition) statement
```

This allows a single statement to be executed or not, depending on the evaluation of a single condition, obviously a limited form of control. It is most often used in conjunction with unconditional branches to model more complex control statements. A more complex form is the IF-ELSE-ENDIF statement used in many languages.

```
IF (condition)
   block1
ELSE
   block2
ENDIF
```

This allows any number of statements to be controlled. And the statements are organized into two blocks, only one of which is executed depending on the evaluation of a single condition. The range of control and the effect of the statement are made clear by the keywords. This form encourages binary thinking. That is, problems are solved by repeatedly dividing them in half. This is not always

the most natural approach.

Another selective control statement is the CASE statement.

```
CASE OF (expression)
  CASE (expression)
    block
  CASE (expression)
    block
  .
  .
  .
  ELSE
    block
ENDCASE
```

This allows the selection of one block of statements from any number of blocks. The selection is made by comparing the first expression to the expressions in each of the following CASE lines. If an equality is found, the associated block is executed and the rest of the statement is skipped. If no equality is found, the block associated with the ELSE is executed. This statement is most comonly used in (and in some languages is restricted to) situations where the first expression is simply a variable, and the subsequent expressions are constants. The conditions being evaluated are then quite simple and the meaning of the statement easily understood.

A common property of all of these statements is that

some default for continued processing always exists if none of the specified conditions is true. Either the statements associated with an ELSE are executed, or if no ELSE is used, the statement is skipped. Thus no matter what the state is on entering the statement, processing always continues after leaving it. This can be very dangerous if some condition has been overlooked, or some invalid input is given.

The original Pascal Case statement which did not include an ELSE block and a more recent proposal by Dijkstra [1976] do not have this property. In Dijkstra's IF statement, blocks of statements are associated with boolean guards. The guards are evaluated in some possibly random order. If one is found to be true, the associated code is executed, and the rest of the statement is skipped. If none of the guards are true, execution is aborted. This means that all possible conditions must be considered. If some unforseen circumstance arises, execution does not proceed along some default path.

3.2 Repetitive Control Statements

A repetitive control statement determines how many

times the statements under its control are executed. This is done by repeatedly executing a statement or group of statements until there is some reason to stop. The simplest form of this is the DO-TIMES statement.

```
DO n TIMES
  block
ENDDO
```

This executes the block of statements as many times as required. Another form is the WHILE or UNTIL loop used in many languages.

```
WHILE (condition)
  block
ENDWHILE
```

This continues to execute the block of statements as long as the condition is true. Once the condition is false, the statements are no longer executed.

These statements consist of two parts. They have a single condition for stopping, and a loop body that is to be repeatedly executed. More flexible versions of this have been proposed. These allow multiple stopping conditions and/or placement of stopping conditions anywhere in the loop body. The ANSI Fortran committee have identified ten variations of the two part loop, and are considering ways of accommodating them in a new standard [Martin 1978], [Meissner 1978], [Wagener 1978].

Unfortunately, a two part loop does not contain or identify all of the components of a repetitive construct. Loop initialization takes place outside of the loop and the iteration mechanism is not identified as a separate component. Even the original Fortran DO statement contained these pieces, although on a very simple level. The initial value, stopping value, and iteration increment may all be specified. However, they are each restricted to be a single integer assignment.

The RATFOR [Kernighan] preprocessor extends the Fortran DO statement in a very nice way. Each of the three components in the loop header is allowed to be a single Fortran statement, including a call to a subroutine. This gives each of the four components a little more power, reducing the need for mixing the iteration mechanism in with the loop body, etc. However, the sequence of execution of the components is still implicit and fixed.

A loop statement that does not fit this pattern is the DO statement of Dijkstra [1976]

```
DO
  (guard) block
  (guard) block
     .
     .
     .
OD
```

Blocks of statements are associated with boolean guards, or conditions. In each iteration, the guards are evaluated, and when a true one is found, its block is executed. This continues until none of the guards are true.

Another way of looking at repetitive control statements is to observe that in many cases, each iteration selects a component from a data structure, and performs some operation on it. Thus the iteration mechanism is related to the data structure. Again, the Fortran DO loop is an excellent example. The fixed integer increment is a very reasonable mechanism for selecting elements from arrays, or rows from matrices. However, even without sophisticated data structuring facilities, it is quite common to model trees or lists using arrays. When this is done, more complex iteration mechanisms are required to select components. This approach has been taken in Alphard [Wulf 1977], which allows specification of an iteration mechanism for user

defined structures. This mechanism is called a generator and includes the initialisation and stopping conditions as well. It is invoked by a special statement and the sequence of initializing, selecting components for each iteration, and testing for termination is fixed.

3.3 Mytran Control Statements

In designing new control statements for this language, the main consideration has been that this is an experimental language. Therefore, it is more important to design something different than to copy a design that has already proven itself. Apart from this, there are also the considerations of clarity, flexibility, and ease of use. The effect of the statement must be easy to understand. The statements must handle a wide variety of situations. And the statements must be a convenient, concise notation for the control function they are performing.

3.3.1 IF Statement

The function of a selective statement, as stated earlier, is to determine which of the statements under its control will be executed. The design used in Mytran is

based on Dijkstra's IF statement. Blocks of statements are associated with boolean expressions.

```
IF
  (KEY GT ENTRY)
     LOW = MID
  (KEY LT ENTRY)
     HIGH = MID
  (KEY EQ ENTRY)
     FOUND = TRUE
FI
```

When the statement is executed, all of the expressions are evaluated. Exactly one of them must be true. The block of statements associated with that expression is executed. If none of the expression are true, or more than one of them is true, execution is aborted.

This statement form allows as many conditions as necessary to be tested at the same level of nesting. The example above would require two levels of nesting using the IF-ELSE statement. The requirement that exactly one alternative be true and the lack of an ELSE block, means that conditions are very carefully specified.

The following example shows how a Fortran IF statement could be written in Mytran.

```
IF (A.LT.B) A=A+1
```

```
IF
  (A LT B) A=A+1
  (A GE B)
FI
```

The extra condition is required since at least one condition must be true or execution will abort. The next example shows how a Pascal IF-THEN-ELSE statement is written in Mytran.

```
If (A LT B)                    IF
   Then A := A + 1               (A LT B) A = A + 1
   Else                          (B LT A) B = B + 1
     If (B LT A)                 (A EQ B) A = 2 * A
       Then B := B + 1         FI
       Else A := 2 * A;
```

Here, the third condition is implicit in Pascal but must be explicitly stated in Mytran. The next example shows how a Pascal Case statement could be written in Mytran.

```
Case I of                      IF
   1 : A := A + 1;               (I EQ 1) A = A + 1
   2 : B := B + 1;               (I EQ 2) B = B + 1
   3 : A := A * 2;               (I EQ 3) A = A * 2
End;                           FI
```

In both of these statements execution aborts if I does not have the value 1,2 or 3.

3.3.2 LOOP Statement

The Mytran repetitive control statement has four parts. They are initialization, stopping conditions, loop body, and iteration mechanism. Each of these consists of a keyword followed by any number of statements. The parts are executed in the order in which they occur in the text and, except for initialization which must come first, the

parts may be used in any order. Any of them may be left out if they are not needed. To allow nesting of loops, the entire statement is bracketed by the keywords LOOP and ENDLOOP.

```
LOOP
  GIVEN block
  WHILE block
  DO block
  LOOPBY block
ENDLOOP
```

The initialization section consists of the keyword GIVEN, followed by any number of statements.

```
GIVEN
  CURRENT = ROOT
  LEN = STRLEN (KEY)
```

If it is used, it must immediately follow the keyword LOOP. It is executed only once, before the repetitive portion of the loop is entered. In fact, the word GIVEN has no effect on execution. The reason it is included is that loops usually require some initialization. This may be assigning a starting value to a loop control variable, or it may be zeroing a matrix, which itself requires a loop. By using the GIVEN block, the initialization is clearly identified as part of the loop. The repetitive portion of the loop begins at the end of the GIVEN block.

The stopping conditions section of the loop consists

of the keyword WHILE followed by any number of statements and conditions.

```
WHILE
  (CURRENT NE Ø)
  ENTRY = TREE (CURRENT,3)
  PTR = STRING (ENTRY,1)
  LEN = STRING (ENTRY,2)
  RESULT = MATCH (KEY,KEYLEN,CHARS(PTR),SLEN)
  (RESULT NE EQUAL)
```

Conditions are boolean expressions enclosed in parentheses. The entire WHILE block is executed sequentially in each iteration and as conditions are encountered, they are evaluated. As soon as one evaluates to false, it causes a branch to the end of the loop. Any number of conditions may be included in a WHILE block and are evaluated independently. In the above example, the two stopping conditions could not be combined into a single one. The calculation of the second one is invalid if the first one is false. If a standard single condition loop were used, additional branches and boolean variables would be necessary. Statements are also allowed as part of the WHILE block. This is because, as in the above example, there are frequently some calculations which are solely related to stopping the loop, rather than producing some result.

The iteration mechanism section of the loop consists

of the keyword LOOPBY, followed by any number of statements.

```
LOOPBY
  IF
    (RESULT EQ LESS)  CURRENT = TREE (CURRENT,1)))
    (RESULT EQ GREATER)  CURRENT = TREE (CURRENT,2)
  FI
```

As with the GIVEN block, the keyword LOOPBY has no effect on execution, though it is part of the repetitive portion of the loop. It merely serves to identify those statements in the loop which are used to "get from" one iteration to the next. In a simple counting loop, this would be the increment or decrement of the loop control variable. In a tree search, as above it is the selection of the appropriate subtree for continued searching.

These three sample blocks give a complete searching loop. It is typical of searching loops that there is no loop body. This is because there is no information being accumulated. All we are doing is getting the next item to look at, and deciding whether or not to stop.

The loop body section consists of the keyword DO followed by any number of statements. These are the statements which actually accumulate a result. Again the keyword DO has no effect on execution. As an extension to

the above examples, if in addition to searching we wanted to keep track of the pathway used, we could add the following.

```
DO PATH (LEVEL) = RESULT
```

Of course, the variable LEVEL would have to be initialised, tested for overflow, and incremented, in the other blocks as well.

The syntax of this statement is somewhat wordy since there are more keywords than are usual in control statements. However, it is a more complete framework within which to design a loop, or determine what a loop is doing.

The following example shows how a Fortran DO loop is written in Mytran.

```
   DO 1Ø I = 1, 5
1Ø A (I) = Ø
```

```
LOOP
  GIVEN I = 1
  DO A (I) = Ø
  LOOPBY I = I + 1
  WHILE (I LE 5)
ENDLOOP
```

One of the common complaints about the Fortran DO loop is that it does not allow Ø iterations. The following example shows how this can be done in a Mytran loop.

```
   DO 1Ø I = J, K
1Ø A (I) = Ø
```

```
LOOP
  GIVEN I = J
  WHILE (I LE K)
  DO A (I) = Ø
  LOOPBY I = I + 1
ENDLOOP
```

The next example shows how a Pascal While loop can be written in Mytran.

```
I := 1;
While (A(I) NE K) DO
  I := I + 1;
```

```
LOOP
  GIVEN I = 1
  WHILE (A(I) NE K)
  LOOPBY I = I + 1
ENDLOOP
```

The loop initialization is contained in the GIVEN block rather than being outside the loop. The Pascal Repeat-Until loop is easily translated into Mytran since the Mytran WHILE block can occur after the DO and LOOPBY blocks.

Chapter IV

Data Types

4.1 Definitions

For the purposes of this discussion the following definitions will be used. A "data type" is a set of values together with operations defined on members of that set. A "data structure" is a member of the set of values of a particular type. A data structure may be an indivisible unit or it may be a collection of items and each item may be an indivisible unit or a further collection of items. For example, an integer is a data type. Its set of values is the set of integers and its operations are integer arithmetic. The number 27 is a data structure of type integer. An array of three integers is also a data type and (3, 6, 2) is a data structure of that type. In programming languages, a variable is a named object which is declared to be of a certain type. It has as its value a data structure of that type. Operations defined for the type are applied to the data structure by using the name of the variable. For example, I and J are integers and their values are 27 and 13. To indicate that these values are to be added

together, an operation which is defined for integers, we write I + J.

Data structures are used to model "real world" objects and situations. As these become more complex it becomes necessary to use more complex data structures and this requires that we be able to describe more complex data types. In order to build these, we need some primitive types, some rules for combining them and a language for describing a type in terms of these primitive types and rules. All programming languages have some ability to build data types, for example in Fortran the primitive types are the scalar types, Integer, Real, Logical, etc. which may be combined as arrays. The language for describing types is the language of the Fortran declaration statements. The Fortran array allows a fixed number of items of a single scalar type to be combined. A typical declaration is

```
INTEGER A (10)
```

This says that the variable A is of type "array of ten integers". Thus we have built a type called "array of ten integers" and declared a variable A to be of that type.

There are two obvious limitations to the types that may be built in Fortran. First, only items of the same

type may be combined and second, only items of scalar type may be combined. There is also a problem with the notation of Fortran declarations, namely that the declaration of variables is mixed in with the description of their types. A more powerful type building language must remove these two limitations and improve on the notation. The first limitation may be removed by allowing items of different types to be combined. The second limitation may be removed by allowing items of any type to be combined. The removal of this limitation implies that recursion is allowed, since the type of an item in a collection may be the same type as the type of the collection itself. This is necessary for the description of recursive types such as trees and lists. Finally, a more natural notation can be achieved by separating type descriptions from variable declarations.

4.2 Pascal Data Types

Pascal is one language that has powerful type building capabilities. The primitive elements for building types in Pascal are, as in Fortran, the scalar types. Items of the same type may be combined into arrays and items of different types may be combined into records. The Pascal type building language allows types to be

described and given a name which may be used in other type descriptions or variable declarations. Items of any type may be combined into arrays or records. A typical Pascal type description is

```
ShortArray = Array [1..50] of Integer;
```

This says that anything of type ShortArray is an array of fifty integers. This can be used to describe a more complex type as in

```
SmallSet = Record
             Size: integer;
             Set: ShortArray
           End;
```

These may be used to declare variables, as in

```
A: ShortArray;
B: SmallSet
C: ^SmallSet;
```

The declaration for C means that it is of type "pointertoSmallSet" and its value will be a pointer to a structure of type SmallSet.

Up to this point we have only considered how to combine types into more and more complex types and have ignored their corresponding operations. Operations fall into four groups as follows: (1) Selectors, which access data structures, (2) Assigners, which replace data structures, (3) Predicates, which carry out tests on data structures and (4) Operators which don't fit into the

first three groups. We now look at some examples of operators and their notation in Pascal. The following are examples of the different types of notation used for selection in Pascal.

```
A
A [4]
B.SIZE
C^
```

The first denotes access to a data structure which is the current value of A. The second denotes access to a data structure which is a single element of the array data structure which is the current value of A. The third denotes access to a data structure which is the value of the SIZE field in the record data structure which is the current value of B. The fourth denotes access to the data structure which is pointed to by the current value of C.

If we use the type and variable declarations from the last examples, then A accesses an array of fifty integers, A [4] accesses an integer, B.SIZE accesses an integer and C^ accesses a record containing an integer and an array of fifty integers. These selectors may be applied repeatedly to access a low level component from a complex structure. For example,

```
C^.SET [4]
```

uses all four types of selector notation to denote access

to an integer.

Assigners in Pascal exactly parallel selectors. Any data structure which can be accessed by a selector can be replaced with another data structure of the same type using an assigner. The notation for assigners is exactly the same as that for selectors and the only distinguishing feature is that assigners appear on the left hand side of an assignment statement and selectors appear on the right hand side. If the notation of the last selector example appears on the right, it causes access to an integer value, but if it appears on the left it causes replacement of the current integer value with a new integer value. This similarity of notation is natural from an implementation point of view since in both cases the translator ultimately generates a machine address, however it confuses the functional point of view since selectors and assigners are clearly very different. A formal treatment of selectors and assigners is given in Six [1980].

Predicates in Pascal are comparisons between two scalar data structures. Any two structures of the same scalar type may be compared for equality or inequality and

if the type is ordered, they may be compared for relative position in the ordering. There are no predicates defined for array or record type structures. The following examples show how the notation is used.

```
A [4] = 6  or  A [4] EQ 6
B.SIZE NE A [1]
C^.SET [1] > Ø
```

In the above expressions, the selectors access the appropriate data structures and then the predicate is applied to them.

Operators in Pascal, outside of those in the first three groups, include binary and unary arithmetic operators and type conversion operators. These are defined only for scalar type structures and not for record or array type structures. These operators produce new data structures which are not simply structural recompositions of their operands. For example, integer addition takes two integer values as operands and produces a new integer value as its result. Binary operators are written in infix notation and unary or type changing operators are written in prefix notation. The following are some examples:

```
I + J
B * C
J DIV 2
CHR (I)
```

where DIV is the integer division operator and CHR is the integer to character type changing operator.

As a summary of the Pascal operators and types that we have looked at, we can say that selectors and assigners are defined for structures of any Pascal type and predicates and other operators are defined only for scalars. Since we have defined data types as a set of values plus operations, we require the ability to define new predicates and operators. This can be done by writing procedures and functions to operate on structures of a particular type. For example, addition is not defined in Pascal for structures of type ShortArray so we may define our own addition operator as a procedure, which has three parameters of type ShortArray and returns the sum of the values of the first two as the value of the third. Thus it is both an addition operator and an assigner. Within this procedure, the addition may be described using any combination of selectors, assigners, predicates and operators that are supported by Pascal. Outside of it, we treat it as an operator which adds two data structures together and produces a third one. Clearly selectors and assigners can also be written as procedures and functions.

4.3 Type Constructors

Flon [1975] describes parameterized types as a specification tool. Parameters may be integers or other types and are used in the description of a new type. For example an integer parameter may be used for the upper bound of an array. Then when this parameter is filled in, an array type of specific size is created. A type parameter may be used as the base type of an array and, when it is filled in, this creates a specific array type. Flon suggests that the use of parameters in type descriptions will give them much wider applicability in the same manner that it does for procedures. He calls them "type constructors" since they may produce widely differing types from different parameters. If we examine this concept, we can see that a type constructor is a rule for combining types to produce new types. In fact, the array is a type constructor, since it is described in terms of parameters which are the upper and lower bounds of its ranges and its base type. Its operators are the selectors and assigners described earlier and it is used to describe types by filling in appropriate parameters as in

```
Array [-4:9] of Integer;
```

The Pascal record is another type constructor and along with the array may be considered to be a primitive

type constructor. With these two, it seems to be possible to construct any type. However, additional type constructors are still needed, both for their notational convenience and the greater degree of abstraction that they allow in type descriptions. New type constructors can be made part of the language definition as additional primitive type constructors, or the language can be given the capability of describing new type constructors. The former approach has been taken in Pascal where the Set and the File type constructors are also included. Each of these may be used with parameters to describe a type and the type described then has selectors, assigners, predicates and other operators associated with it. Adding new type constructors to a language definition is not a solution since it does not give extensibility. A much more general solution is achieved by providing the capability for describing new type constructors.

For example, a tree is a type constructor. Its nodes can be described as having keys, information and pointers to other nodes. The pointers and their manipulation can be described fully, but the type of key and the type of information are parameters. When a specific tree type is described, these parameters are filled in with appropriate types, such as in the type "a tree with integers for keys

and ten-character strings for information".

4.4 Mytran FORMs

In order to describe new type constructors, we need some primitive types, some primitive constructors and a notation in which to express the description. Mytran is a language which supports the description of type constructors. Its primitive types are Fortran scalars, and its primitive constructors are the array and record. Type constructors are defined as FORMs which are based on Flon's [1975] parameterized types. The notation and capability of a FORM is shown in the following examples.

A FORM for a stack may be defined as follows,

(1)
```
FORM INTSTACK (MAX : INTEGER)

    RECORD
       PTR : INTEGER
       ST : ARRAY (MAX) OF INTEGER
    ENDREC

    SUBROUTINE PUSH (VALUE)
    VALUE : INTEGER
    BEGIN
      IF (PTR GE MAX) STOP "STACK OVERFLOW"
         (PTR LT MAX) PTR = PTR + 1; ST (PTR) = VALUE
      FI
    RETURN
    END


    SUBROUTINE POP (VALUE)
    .
```

```
        .
        .
    ENDFORM
```

The declaration

```
(2)  S : STACK (100)
```

will create a variable S which will be of type "stack of 100 integers". The value of S will be a record whose fields may be accessed using the same selector notation as for Pascal records. In addition the operators PUSH and POP are defined for the value of S and the notation for invoking them is

```
(2)  CALL S.PUSH (I)
     CALL S.POP (I)
```

Another declaration

```
(4)  T : STACK (50)
```

will create a variable T of the new type "stack of 50 integers". To invoke operations on the value of T, the notation is

```
(5)  CALL T.PUSH (I)
     CALL T.POP (I)
```

A more general stack form may be defined as follows

```
(6)  FORM STACK (MAX:INTEGER, ENTRY:FORM)

       RECORD
         PTR : INTEGER
         ST : ARRAY (MAX) OF ENTRY
       ENDREC

       IMPORT FROM ENTRY ASSIGN (E1, E2)
```

```
SUBROUTINE PUSH (VALUE)
VALUE : ENTRY
BEGIN
  IF (PTR GE MAX) STOP "STACK OVERFLOW"
     (PTR LT MAX) PTR = PTR + 1
       CALL ST (PTR).ASSIGN (VALUE, ST (PTR))
  FI
RETURN
END

SUBROUTINE POP (VALUE)
.
.
.

ENDFORM
```

The type of information held in the stack is now a parameter so any type of stack may be built with this constructor. An ASSIGN subroutine is imported from this parameter, so any type which is used in a STACK description must export an ASSIGN subroutine. Imported functions and subroutines are the only way that a FORM can describe operations on components whose types are parameters. In the above example, a stack must be able to store and return information via the PUSH and POP operators. However the nature of this information is unknown, so no precise definition of how to copy it can be given. Instead the FORM requires that whatever type is used must support a subroutine to do this copying. So for example, the declaration

(7) S : STACK (100, INTEGER)

is not valid, since the Fortran type INTEGER has no

subroutines associated with it. However, a trivial form may be defined

```
(8)  FORM INTYPE INTEGER

       EXPORTS INTASGN AS ASSIGN

       SUBROUTINE INTASGN (I1,I2)
       I1, I2 : INTEGER
       BEGIN I2 = I1
       RETURN
       END

     ENDFORM
```

And now the declaration

```
(9)  S : STACK (100, INTYPE)
```

is valid, and creates a variable S of the same type as in the first example. Obviously, this is a very messy way to implement a stack of integers. However, if the following FORM is defined

```
(10)  FORM STRING (MAX : INTEGER)

        RECORD
          LEN : INTEGER
          STR : ARRAY (MAX) OF INTEGER
        ENDREC

        EXPORTS STRCPY AS ASSIGN

        SUBROUTINE STRCPY (S1,S2)
        .
        .
        .

      ENDFORM
```

Now the declaration

```
(11)  S : STACK (100, STRING (5))
```

creates a stack of 5-integer strings, without rewriting

any of the STACK routines. Assuming the declaration

(12) STR : STRING (5)

then the operation S.PUSH (STR) will push the string in STR on top of stack S.

A FORM description has four parts, the header, the structure, the import/export lists and the procedures. The general syntax is

```
FORM name (parameters)
     structure description
     IMPORT import list
     EXPORT export list
     procedures
ENDFORM
```

The header contains the name of the FORM and any parameters. Parameters may be either types or integers. They are used in the structure description and in the procedures. Integer parameters are typically sizes. In example (1), the header is

INTSTACK (MAX : INTEGER)

MAX is the size of the stack, and is used as the dimension for the array that holds the stack, and to test for overflow in a procedure that adds items to the stack. Type parameters are used to allow more general FORMs to be described. In example (6), the header is

STACK (MAX:INTEGER, ENTRY:FORM)

MAX is again the size of the stack, and is used as before.

The parameter ENTRY is the type of the information to be kept on the stack. Thus the FORM STACK is described without regard to the nature of the information that it is to hold. As in example (11), it is quite easy to describe a stack of strings, and if a general string form is described, to describe a string of stacks.

The structure description describes the other type constructors and types being used. There are five types of structure description, record, array, Fortran, form and form parameter. Record descriptions are the same as Pascal records. The syntax is

```
RECORD
  FIELD1 : structure
  FIELD2, FIELD3 : structure
ENDREC
```

The structure of a field may be any one of the five structure types.

Array descriptions are almost the same as Pascal. The syntax is

```
ARRAY (rangelist) OF structure
```

The rangelist is a list of ranges separated by commas. A range may be given as either a lower and upper bound separated by a colon or an upper bound in which case the lower bound is assumed to be 1. Bounds may be integer

constants or integer parameters. The structure of array elements may be any of the five structure types.

Fortran descriptions are just the Fortran scalar types, INTEGER, REAL, LOGICAL, etc. Form structures are previously defined FORMs, with all necessary parameters. The structure in example (2)

```
STACK (100)
```

is a form structure description.

Form parameter structure descriptions are simply type parameters. The base structure of the array ST in example (6) is ENTRY which is a parameter to the FORM STACK.

The Import/Export lists describe the interface between FORMs. The Import list describes the operations that must be supported by any type parameters. These are procedures and functions that are used in procedures to manipulate information in the FORM. In example (6), the subroutine PUSH must be able to transfer information from the variable VALUE into the stack. Since VALUE is of type ENTRY, a parameter, there is no way to describe the transfer. So a subroutine ASSIGN is called to do it. This subroutine is imported from the FORM parameter ENTRY. Thus any type which is used as an actual parameter in

describing a stack must export a subroutine ASSIGN.

export list describes operations that are supported by a FORM. Subroutines and functions may be exported via alias names. A routine may be exported by more than one alias name. In example (10), the subroutine STRCPY copies strings. It is exported under the alias name ASSIGN.

The procedures for a FORM describe the operations that may be performed on a structure of any type constructed by the FORM. They are Fortran subroutines and functions and may have parameters whose types are parameters to the FORM. In example (6) the VALUE parameter is of type ENTRY which is a parameter to the FORM. Integer parameters to the FORM may be referenced in the subroutines and functions of the FORM, typically for checking bounds and overflows, as in example (6).

4.5 Other Languages

There are a number of modern languages which provide type building capabilities. We will now look at some of them and compare them to Mytran. Although we only discuss their data type facilities, they all have many other

features to recommend them. A summary of their features is given in Hanson et al [1979]. We will describe their capabilities in terms of Pascal data types because they are the most widely known.

The first level above Pascal types is the class of Simula [Birtwhistle et al 1973] and Concurrent Pascal [Brinch Hansen 1977]. This binds together a set of local objects and procedures and functions that operate on them. Variables are declared as instances of these classes and information held by a variable may only be manipulated by the operations of its class.

A different approach on the same level of complexity is the module. This goes under different names in different languages but generally it is a means to bind together a set of objects and some procedures and functions in the same way as in a class. The difference between them is that a class is a type and variables are created as instances of that type, whereas a module is an information hiding mechanism, within which can be defined any number of types and operations on them. Although a variable may be declared to be of a type that is defined in a module, and therefore operations in the module may be used on it, the module itself is not a type. Languages

that use the module concept, such as Modula [Wirth 1977] and Euclid [Chang et al 1978], support the view that processes consist of independent, communicating subprocesses which have exclusive control of their own data. This is a much broader concept than that of data type, but it does encompass some aspects of it. A data type may be described in a module complete with its operators. Modules and classes at this level are fixed specifications with no parameters. Although several instances of a class may be created, they all have the same properties. From the point of view of type building the only extension offered is that operations are included with the rest of the type description.

The next level of languages allow modules or classes to have parameters which may themselves be modules or classes. CLU [Liskov et al 1977], Alphard [Wulf 1977] and an unnamed successor to Pascal [Robinson 1980] are languages with this capability. Mytran is also at this level. In addition, CLU and Alphard allow overloading of operators. This means that an operator supported by the language may be given a new meaning in a type description. For example, in the description of a matrix, the operator + can be defined to perform matrix addition.

A further capability is supported in Alphard, Mesa [Geschke et al 1977] and Modula-2 [Wirth 1980], namely separate definition and implementation modules. This means that the properties of a data type may be defined separately from their implementation. This has two advantages. One is that other programmers who wish to reference a module need only see the definition module. The second is that if an implementation module is changed, a certain amount of consistency checking can be done against the definition module. Although this does not extend the capability to describe types, it does make them easier to use and much more secure.

These last two features are not included in Mytran though they could be added. A more basic difference between Mytran and all of these languages is the manner in which storage is allocated for data structures. This is admittedly a problem of implementation rather than design, but Mytran was designed so that it could be easily implemented in Fortran.

These languages all use a stack based dynamic storage allocation scheme. When the execution of a module is initiated, the storage required for its structures is allocated and variable names are bound to their

structures. Since a module is a coroutine, control may leave it without terminating it, so its structures remain active, though hidden from other modules. If another instance of the same module is initiated, storage for a new set of structures is allocated. Mytran uses Fortran as its object language and therefore has static storage allocation. The space needed for a variable is calculated from its declaration at translation time and storage is statically allocated. Also, since Fortran is the object language, the values of variables local to a subprogram are not active once control has left that subprogram.

Chapter V

Implementation

This chapter discusses the implementation of the translator. An outline is given of implementation problems, their solutions, and the implementation strategy. There are also a number of examples of translations of program or statement fragments. A complete sample translation is given in Appendix A.

Mytran was implemented in two stages. The first stage added the new control statements IF and LOOP. It was written in itself and hand compiled. The second stage added data structuring via the FORM. It was written using the stage one language.

5.1 First Stage

This stage added two new control statements to Fortran, the IF statement and the LOOP statement. The Fortran IF and DO statements were dropped. The translator is a statement preprocessor as described in Chapter I. It

looks for key words or symbols and performs a fixed translation on them. Everything else is copied. The result is a complete language in the sense that all Fortran statement types are either copied or have an equivalent in the new statements. For example Fortran Read and Write statements are accepted as source and copied unchanged. Fortran IF and DO statements are not accepted but their function is replaced by the Mytran IF and LOOP statements. Thus any program which may be written in Fortran, may also be written, hopefully more easily, in Mytran.

The translator also allows multiple statements per line. The need for this follows naturally from the new control statements. First of all the Loop statement is so wordy that it is a great help in simple cases to put several parts of it on one line. More importantly because these statements have a nested structure, that is the range of control of an IF or LOOP statement may be any number of statements, the notion of one statement per line no longer makes sense. However, end of line is still considered a statement terminator unless the following line is a continuation line. The Fortran source format for continuation and labels is also retained. These rules improve readability. In particular if labels must be used, they should be visible.

5.1.1 IF Statement

The following is a typical Mytran IF statement.

```
1   IF
2     (KEY .EQ. ID)
3       FOUND = .TRUE.
4     (KEY .LT. ID)
5       HIGH = MID
6     (KEY .GT. ID)
7       LOW = MID
8   FI
```

The semantics of this statement, as described in Chapter III, say that all three of the conditions will be evaluated simultaneously and the statement that is associated with the one that is true will be executed. If none of the conditions is true, or more than one is true, then execution will be aborted. The first observation about this with regard to implementation is that the evaluation of conditions cannot really take place simultaneously, so there will have to be some way of simulating this via sequential evaluation. This can be done by evaluating all the conditions in some order and keeping track of those that are true. After all the conditions are evaluated, if exactly one is true, the statements associated with it are executed. If the restriction is made that none of the conditions has side effects then the sequence of evaluation doesn't matter.

This cannot be checked by the translator without eliminating functions calls from conditions, and so it must be a programmer discipline.

We can now identify some of the requirements of the implementation. First, it must evaluate all conditions of an IF statement. Second, if a condition evaluates to true, there must be some way of transferring control to its block of statements after all conditions have been evaluated. Third, if a previous condition was true execution must be halted with a suitable error message. This may be done immediately upon finding the second true condition or after evaluating all conditions. Fourth, after evaluation of the last condition, either control must be passed to the block of statements associated with the true condition, or if none is true, execution must be halted with a suitable error message. These error messages should give the line numbers at which the errors occurred. And fifth, after execution of a block of statements control must be transferred to the statement following the end of the IF statement. There are also two overall requirements. First, translation should be done one statement at a time with a minimum of information retained by the translator between statements. Second, translation of nested control statements must not require any special treatment.

To implement the first requirement we do the following. If a condition is false branch immediately to the next condition. If it is true set any necessary flags and pointers and then branch to the next condition. To satisfy the second through fourth requirements the conditions in an IF statement are numbered sequentially starting at 1 and if a condition is true a local system variable, LABEL is given the condition's number. This can be used as an index to get to the appropriate block of statements after all conditions are evaluated. If LABEL has been set previously then two conditions are true and execution can be halted. And if after evaluating all conditions LABEL has not been set then no condition is true so execution can be halted. If a condition is true its line number is recorded so it may be reported in any error messages. The fifth requirement is satisfied by branching at the end of each block to a point past the end of the IF.

Here is a complete translation of the example using the above strategies, with source on the left and generated code on the right.

```
1  IF                          LABEL=0

2     (KEY .EQ. ID)            IF(.NOT.(KEY.EQ.ID))GOTO 20
                               CALL TEST01(LABEL,LINE1,2,1,NAME)
                               GOTO 20
                            30 CONTINUE

3       FOUND = .TRUE.         FOUND=.TRUE.

4     (KEY .LT. ID)            GOTO 10
                            20 IF(.NOT.(KEY.LT.ID))GOTO 40
                               CALL TEST01(LABEL,LINE1,4,2,NAME)
                               GOTO 40
                            50 CONTINUE

5       HIGH = MID             HIGH=MID

6     (KEY .GT. ID)            GOTO 10
                            40 IF(.NOT.(KEY.GT.ID))GOTO 60
                               CALL TEST01(LABEL,LINE1,6,3,NAME)
                               GOTO 60
                            70 CONTINUE

7       LOW = MID              LOW = MID

8  FI                          GOTO 10
                            60 IF(LABEL.EQ.0)CALL ERR001(1,NAME)
                               GOTO(30,50,70),LABEL
                            10 CONTINUE
```

Code is generated for each component of the IF statement as follows. When the keyword IF is found, generate

LABEL = 0

This initialises the system variable which records the number of a true condition. At the same tiime, a label is generated for the end of the IF statement and the condition counter for this IF is set to zero. When the first condition is found, generate

```
      IF (.NOT.(KEY.EQ.ID))GOTO 2Ø
      CALL TESTØ1(LABEL,LINE1,2,1,NAME)
      GOTO 2Ø
   3Ø CONTINUE
```

The label 2Ø is generated for the next condition, and the label 3Ø is generated as the start of the associated block of statements. If this condition is the only true one, a branch will be made to this label. TESTØ1 is a system routine that sets the necessary flags and indices for a true condition. It is given the system variable LABEL which holds the number of a true condition, and LINE1 which holds the line number of a true condition. The line number of this condition, 2, is also given as is the condition number, 1. The system variable NAME holds the name of the current subroutine and will be used if an error is found.

When the next condition is found, generate

```
      GOTO 1Ø
   2Ø CONTINUE
      IF(.NOT.(KEY.LT.ID))GOTO 4Ø
      CALL TESTØ1 (LABEL,LINE1,4,2,NAME)
      GOTO 4Ø
   5Ø CONTINUE
```

The first line is generated because this condition indicates the end of the block of statements associated with the previous condition and if this block of statements is executed it must end by branching past the end of the IF statement. The label 1Ø has been allocated for this purpose. The second line is generated because the previous

condition must be able to branch to this one. The rest is generated as for the first condition and all subsequent conditions are generated in the same manner as this one.

When the keyword FI is found, generate

```
      GOTO 1Ø
   6Ø CONTINUE
      IF (LABEL.EQ.Ø)CALL ERRØØ1(1,NAME)
      GOTO(3Ø,5Ø,7Ø),LABEL
   1Ø CONTINUE
```

The first line is generated to terminate the previous block of statements. The second line is generated so that the last condition can branch to here. All conditions will have been tested at this point, so the third line is generated to check that one of them was true, and if not, to call an error routine. The next line is generated to cause a branch back to the appropriate block of statements. And the last line is the point to which each block of statements branches.

This translation clearly works on a line by line basis, and the translator only needs to retain several label values and the condition counter between lines of the statement. The fact that it works for nested IF statements is slightly more subtle. The key point is that any nested IF statement is part of a block of statements and therefore cannot be encountered until all the conditions of the

current level have been tested and are finished with. At this point all of the system variables such as LABEL are free to be reused.

5.1.2 LOOP Statement

The following is a typical LOOP statement

```
LOOP
  GIVEN I = 1
  WHILE (I .LE. LIMIT)
  DO A (I) = Ø
  LOOPBY I = I + 1
ENDLOOP
```

The semantics of this statement as described in Chapter III say that the statement in the GIVEN block will be executed once on entry to the LOOP statement and the statements in the remainder of the blocks will be repeatedly executed until a condition in the WHILE block evaluates to false. This translation is very straightforward. There are the following requirements. There must be a branch from the end of the loop to the start of the repetitive part of the loop, in this case the WHILE block. If a condition in the WHILE block evaluates to false it must cause a branch to the first statement after the end of the LOOP statement. As can be seen from the following translation of the example, most of the keywords are simply discarded and cause no code to be

generated.

```
      GIVEN I = 1                          I=1
      WHILE (I .LE. LIMIT)              1Ø IF(.NOT.(I.LE.LIMIT))GOTO 2Ø
      DO A (I) = Ø                         A(I)=Ø
      LOOPBY I = I + 1                     I=I+1
    ENDLOOP                                GOTO 1Ø
                                        2Ø CONTINUE
```

5.2 Second Stage

This stage added data structuring via the FORM. FORMs are a major extension to Fortran data declarations and so it was necessary to build a complex symbol table from the source. There is an equally major extension to the manner in which data may be referenced. This necessitated the parsing of source statements into operands and operators, and parsing operators into reference components. Many of the usual parts of a compiler are present in the translator though since it is generating Fortran rather than machine code they are simpler.

5.2.1 FORMs

The following is part of a typical FORM.

```
FORM

  SET (ELEMENT:FORM, MAX:INTEGER)

  RECORD
    A : ARRAY (MAX) OF ELEMENT
    SIZE : INTEGER
  ENDREC

  IMPORT FROM ELEMENT
                EQUAL (X,Y):LOGICAL
                COPY (X,Y)

  EXPORT SETMEM AS MEMBER
         SETADD AS INSERT
  .

ENDFORM
```

A typical declaration using this FORM is

```
X : SET (ENTRY,50)
```

The semantics of this as described in Chapter IV say that X is a structure containing 50 occurences of a structure of type ENTRY which has been previously defined and one integer. The components of X are referenced as X.A, which refers to an array of 50 items of type ENTRY, X.A (I) which refers to a single item of type ENTRY, and X.SIZE which refers to a single integer. Any components defined in the type ENTRY may be referenced by adding the necessary information to the end of X.A (I). For example if ENTRY is an array of integers then X.A (I) (J) will

refer to a single integer, or if ENTRY is a record with field P of type integer, then X.A (I).P refers to a single integer. Also the type ENTRY must export a function under the name EQUAL and a subroutine under the name COPY. Any routines in SET that reference EQUAL or COPY will, if invoked via X, actually reference the exported function or subroutine.

When the FORM SET is encountered in the source there no way of knowing that it will be used in conjunction with ENTRY. Thus there is no way of knowing how big a structure of FORM SET will be. In particular there is no way of knowing how to access the components of a SET since their size is unknown. On the other hand routines within SET must be able to describe operations on these components at least in a general manner. The first observation to make is that in fact these routines may only be invoked via a variable such as X in the example which has been declared to be of FORM SET. This declaration will contain all the necessary parameters to solve the referencing problem. It remains only to determine a means of communicating this information to the FORM. There are several possibile methods which I will call the macro method, the interpretation method, and the dope vector method.

5.2.1.1 The Macro Method

This method involves treating FORM routines as macros which are expanded when a declaration is encountered. This has been suggested by several people [Gries 1977], [Holt 1979] and is used in the language Model [Morris 1979]. Model expands small procedures in line and generates new copies of larger ones for each new declaration. This is comparable to generating index calculations for a Fortran array as inline code. This is a valid technique for a simple situation such as this. However user defined access mechanisms may be arbitrarily complex so there is the possibility of an explosion of duplicated code.

5.2.1.2 The Interpretation Method

This method involves building a symbol table for each FORM and each declaration. The declaration symbol table for a variable is passed to any FORM routines accessed via that variable. To access a component of a FORM, a system routine is called at execution time and passed the declaration symbol table for the current variable, the FORM symbol table, and the component name. The system routine

searches for the component name in the FORM description. If a form parameter is encountered its value is obtained from the declaration symbol table. This method leaves the bulk of the referencing work to execution time. As with the macro method it may be acceptable for simple structures but it is not a good general solution.

5.2.1.3 The Dope Vector Method

In this method as much of the work as possible for calculating a reference is done at translation time. Any sizes that are available are generated in a reference expression. Any that are not available are assumed to be in a dope vector at a specific address. When a variable is declared all of the unknown information will be available and the necessary values are put into the dope vector. Whenever a FORM routine is accessed via this variable its dope vector must be available to the routine. There are two ways of doing this. The first is to keep the dope vector in the structure. The second is to keep the dope vector outside of the structure. The first is the most powerful and convenient. If all structures carry their own descriptions they can never be misused. However components of structures must also contain their own descriptions and this obviously leads to an explosive proliferation of

descriptions. The second method is not quite so convenient since a structure cannot be understood without its dope vector, and it may be misinterpreted if the wrong dope vector is used. However dope vectors are completely internal to the system. They are generated by the translator so it is possible to ensure that the correct one is always used.

5.2.2 Implementation of Dope Vectors

The dope vector method was chosen to implement Mytran FORMs. The general implementation strategy is as follows. When a FORM is scanned, a symbol table is built containing any access information that is in the FORM. Also locations are allocated in the dope vector for any information that is not available but will be needed for references. References within the FORM are generated in terms of the known information and the appropriate dope vector locations. When a variable is declared using the FORM a symbol table is built for it which is a copy of the FORM symbol table, expanded to include the new information that is in the declaration. For example the FORM symbol table for SET will say that the base type of the array A is a parameter so its size is unknown. Any access expressions involving this size will generate a reference to the dope

vector. When the declaration for X is found a symbol table for it will be created. This will say that the base type for array A is the type ENTRY and a complete description including its size will be given. As this table is being built all the information that was previously unknown is put into the dope vector for use at execution time. Any references to components of X will generate access expressions entirely in terms of known sizes.

The dope vector is a convenient mechanism for communicating information about a specific declaration to a general FORM. The information in the dope vector is determined by the requirements of the access expressions which are generated for references.

5.2.3 References

In Mytran, structures are stored linearly. Regardless of the hierarchical nature of a logical structure it is mapped onto a linear physical representation. A reference to a variable or a component of a variable is a reference to the start of storage of the variable or component. For example given the FORM SET and declaration X at the start of this chapter, and assuming that the type ENTRY as an array of 10 integers, then a reference to X is a reference

to the first word of X, and a reference to X.A is a reference to the first word of X. A reference to X.A (4) is a reference to the 31st word of X and a reference to X.SIZE is a reference to the 501st word of X. The translator must accept logical references of this nature and translate them into the appropriate physical references. As discussed in the last section, not all of the necessary information is available at translation time, so the translator must actually generate access expressions which will be evaluated at execution time.

Since Mytran is generating Fortran object code, a declaration of a variable with hierarchical structure is translated into a declaration of an appropriate length one-dimensional Fortran array. Access expressions are subscript expressions accessing the appropriate word in the array.

There are two language supported selectors in Mytran. They are the record and the array. Access to elements of arrays is done by calculating the element number within a linear address space and multiplying this by the size of each element. This gives the offset from the start of the structure to the desired element. Access to fields of records is done by adding up the sizes of all preceding

fields. This again gives an offset. There are several observations that may be made at this point. The first is that several calculations are involved in changing a set of indices into an element number and each index is individually checked to ensure that it is within its ranges. This can be most conveniently done by passing the indices and a list of ranges to a function which will perform the necessary operations. But this means that the range list might as well be in the dope vector. The second observation is that the size of the base element of an array is needed, but may not be known within a form due to parameterization. If this is the case, it must be put in the dope vector. The third observation is that a field reference always refers to a fixed location, so that the sum of the preceding fields may be calculated at translation time allowing a field to be accessed by a single number at execution time. Again if the form of some field is a parameter then the offsets for all following fields are unknown within the current form so they must come from the dope vector. The final observation is that these access mechanisms may be applied sequentially to reach a low level component of a complex structure.

5.2.4 Implementation of References

The implementation strategy is as follows. When a reference to a variable is encountered, its symbol table entry is found. If there are several levels of specification in the reference the translator "walks through" the structure description in the symbol table. As a particular component is specified at each level the translator uses information from the symbol table about that component to generate an access expression. The following are some sample references and their translations based on the form SET and the declaration X.

(1)	X.A	X(Ø+1)
(2)	X.SIZE	X(5ØØ+1)
(3)	X.A (I+J)	X(Ø+IX1(LOCAL(n),I+J)*1Ø+1)

Access expressions are calculated from a base of zero, but Fortran arrays are addressed from a base of one, so the translator adds one onto the end of each access expression. In (3), IX1 is the system function that calculates the offset and checks the range of indices for a one-dimensional array. LOCAL is the local dope vector and n is the start of the range list for the array. 1Ø is the size of the form ENTRY. If a declaration of a variable of form SET accurs within the form, its parameters are unknown, as in the following

```
Y : SET
```

Access expressions for Y must now be generated in terms of unknown information.

```
(1)  Y.A                 Y(Ø+1)
(2)  Y.SIZE              Y(TYPE(m)+1)
(3)  Y.A (I+J)           Y(Ø+IX1(TYPE(n),I+J)*TYPE(p)+1)
```

In (1) the offset to field A does not rely on any parameters and so is known at translation time. In (2) the offset to field SIZE is not known since the size of the preceding field is based on a parameter. TYPE is the dope vector which will be passed for a specific variable of form SET and m is the location allocated to hold the offset of field SIZE. In (3) IX1 is as described above, n is the start of the range list though here the range list has been passed in TYPE rather than being local and p is the location allocated to hold the size of the base element of array A.

5.2.3 Imported Subroutines and Functions

There is one remaining problem in generating code for FORMs. This is related to calling imported subroutines and functions. Just as there is no way of knowing at the time a form is scanned how big a SET will be, there is also no way of knowing how to call the function EQUAL and the subroutine COPY. These are aliases and all that is guaranteed is that whatever form is used for the parameter ELEMENT it will export something under these aliases. As discussed earlier one solution is to treat forms and their

routines as macros. When a call to COPY is expanded the appropriate actual name can be generated. This generation of duplicate code is unacceptable for the reasons stated earlier, so another solution is required.

The method used in Mytran is to keep a table of exported routines containing the form from which it was exported, its exported alias and its actual name. Then when a reference is made to an imported routine, it is translated to a reference to the routine via its actual name. Note that this translation takes place at execution time, so that the desired reference is actually to the address of the routine. The most efficient implementation would be to have a table of subroutine and function addresses and by selecting the correct address, call the correct routine. These addresses could be held in the dope vector. Due to differences between machines in handling subroutine and function calls, this cannot be done in Fortran. As a result Mytran generates "alias routines" which contain calls to all routines exported under each alias. So when the form SET calls the subroutine COPY, it actually calls an alias routine which then calls the appropriate routine depending on the parameters used in the declaration of the variable through which SET has been accessed.

Chapter VI

Conclusions

To determine whether this project succeeded or failed let us review the goals and the extent to which they were met.

6.1 Control Statements

The first goal was to design and implement some new control statements. This resulted in the IF and LOOP statements described in Chapter III. These statements were implemented in the first stage of the project, which was written using these statements and then hand translated. The second stage was written using the stage one translator. In all, about 7000 lines of code have been written using these control statements. They were adequate for all situations that occurred and were in some respects superior to other control statements. The following comments are based on this experience.

6.1.1 IF Statement

As can be seen in the examples in Chapter III, the IF statement requires a certain amount of redundancy in specifying conditions. Even if there is only one condition which leads to an action, the opposite condition must still be specified with no action. This redundancy has two benefits. The first is that typing errors are trapped since exactly one condition must be true. This is exactly the type of error that turns into a persistent bug unless it is caught early. The second is that there is a sense of completeness to a selective control statement when it is known that all possible conditions have been tested. Even if some of them lead to no action, it is clear that the condition was not overlooked but rather an explicit decision was made to do nothing. The price to be paid for this is that these redundant conditions must be written and this can be very tedious.

One gratuitous effect of the IF statement is that the threat of execution being aborted encourages careful programming. If a number of complex conditions are specified it may not be clear whether all possibilities have been covered, or whether more than one of them could be true. Rather than risk a fatal execution error, it is

easier to break them up into simpler conditions. This means that the code is much easier to understand at a later review.

6.1.2 LOOP Statement

The LOOP statement is an entirely new design. No language that I have seen uses a repetitive control statement that clearly identifies the four components of a loop. As can be seen in the examples in Chapter III, a LOOP easily models many other repetitive constructs. The fact that multiple, independently tested stopping conditions are allowed and that they may be placed anywhere in the loop body, removes the need for additional boolean variables to keep track of stopping conditions. However its most important function is to provide a framework within which to describe a loop. This is an advantage both when a program is being written and when it is being read at some later date.

One feature that could be added is a Next Iteration statement. This would cause an immediate branch to the start of the LOOPBY section to generate the necessary state for the next iteration. Unfortunately this would require some rules concerning sequence of execution of the

components to ensure that the WHILE section was not skipped.

6.2 Data Structures

The second goal was the support of data abstraction. This resulted in the design of parameterized types or type constructors called FORMs as described in Chapter IV. These were implemented in the second stage of the project and have not been used except for some small examples. Therefore no very significant comments can be made concerning their effectiveness. The only available measure of success is a comparison with other languages. As noted in Chapter IV, Mytran is missing several features that are useful for data abstraction. But on the central issues it is acceptable. Most writers say that the most important consideration is the separation of implementation from usage. The FORM is an adequate mechanism for achieving this. Although there is no enforced hiding of information local to a structure, it is easy enough to do this through programming discipline. Of even greater importance, in my opinion, is the ability to define several simple objects in general terms, and then combine them into an object that is far too complex to be easily described as a whole. The Mytran FORM supports this at least as well as any other

language.

6.3 Portability

The third goal was to have a portable system. This resulted in using a Fortran preprocessor for the implementation. Portability cannot be claimed until it has been done. Although every attempt was made to use widely supported Fortran, there is no way to prove that the translator can be moved, without moving it.

6.4 Future Work

There are number of interesting possibilities for future work. The problems of I/O and literals for structured data have been ignored in Mytran, as they have in most languages. The language PPL [Wallis 1980] is one that contains some facilities for defining I/O and literal formats though only for unparameterized structures. It may be possible to extend these ideas for the parameterized Forms of Mytran, or it may be necessary to design new facilities. Another possible project is to add user definition of infix operators for Forms. As discussed in Chapter IV this is absolutely necessary to achieve independence of data structures from algorithms. A related

project is to allow functions to return non-scalar structures. These two features would allow many algorithms to be written in a much more natural manner.

If any future work is to be done on Mytran, the decision to implement it as a Fortran preprocessor should be reviewed. Although anything can be done in Fortran, there are some things which cannot be done easily. In particular any modern ideas about nested scopes of variables cannot be easily translated into Fortran. Also recursion and dynamic storage allocation, although not difficult on an ad hoc basis, are quite difficult to translate in general. Thus another possible project would be to change the Mytran preprocessor to a compiler. The structure of the translator is such that generation of machine code would not be difficult.

Bibliography

Arisawa, Makoto, Minoru Iuchi (1979) "Fortran + Preprocessor = Utopia 84", Sigplan Notices 14,1 Jan/79,p.12-16.

Barnard, David T., W. David Elliott, David H. Thompson (1979) "Euclid and Modula", Sigplan Notices 13,3 May/78, p.70-84.

Bauer, F. L. (1974) "Compiler Construction: An Advanced Course ",Springer-Verlag,Berlin,1974.

Birtwhistle, G. M., O-J. Dahl, B. Myrhaug, K. Nygaard (1973) "Simula Begin ",Auerbach Publishers Inc., Phila.

Brinch Hansen, P. (1977) "The Architecture of Concurrent Programs ", Prentice-Hall Inc., Englewood Cliffs, N.J.

Brown, P. J. (1974) "Macro Processors ", Wiley, London.

Campbell-Kelly, M. (1973) "An Introduction to Macros ", Macdonal/American Elsevier, N. Y.

Chang, Ernest, Neil E. Kaden, W. David Elliott (1978) "Abstract Data Types in Euclid", Sigplan Notices 13,3 Mar/78, p.34-42.

Cheatham, T. E., Judy A. Townley (1974) "A Proposed System for Structured Programming ", Harvard Univ., Ma.

Cook, A. James, L. J. Shustek (1975) "A User's Guide to Mortran2 ", Computer Research Group, Stanford Linear Accelerator Centre.

Crary, F. D. (1974) "The Augment Precompiler ", MRC Technical Summary Report #1469, University of Madison-Wisconsin.

Dijkstra, E. W. (1976) "A Discipline of Programming ", Prentice-Hall,N.J.

Flon, Lawrence (1975) "Program Design with Abstract Data Types ", Dept. of Comp. Sci., Carnegie-Mellon University.

Geschke, Charles M., James H. Morris Jr, Edmund H. Satterthwaite (1977) "Early Experience with Mesa", CACM 20, Aug/77, p.540-553.

Gries, David (1971) "Compiler Construction for Digital Computers ", Wiley, N. Y.

Gries, David, Narain Gehani (1977) "Some Ideas on Data Types in High Level Languages", CACM 20 June/77, p.414-420.

Hanson, S., R. Jullig, P. Jackson, P. Leng, T. Pittman (1979) "Summary of the Characteristics of Several Modern Programming Languages", Sigplan Notices 14,5 May/79, p.28-45.

Hoare, C. A. R. (1972) "notes on Data Structuring", Structured Programming , Academic Press, London.

Holt, Richard C. David B. Wortman (1979) "A Model for Implementing Euclid Modules and Type Templates", Sigplan Notices 14,8 Aug 79, p8-12.

Jensen, K., N. Wirth (1975) "PASCAL: User Manual and Report, 2nd ed. ",Springer-Verlag, N.Y.

Kernighan, Brian W. "Ratfor - A Preprocessor for a Rational Fortran ",Bell Laboratories, N. J.

Ledgard, Henry F., Robert W. Taylor (1979) "Two Views of Data Abstraction", CACM 20 June/77, p.382-384

Liskov, B., Allan Snyder, Russell Atkinson, Craig Schaf (1977) "Abstraction Mechanisms in CLU", CACM 20 Aug/77, p.564-576.

Martin, Jeanne T. (1978) "Looping Structures", Minutes of 64th Meeting , X3J3/106, ANSI, p.79-88.

Meissner, Loren P. (1978) "Control Structure Extension", Minutes of 65th Meeting , X3J3/108, ANSI, p.55-65.

Morris, James B. (1979) "Data Abstraction: A Static Implementation Strategy", Sigplan Notices 14,8 Aug/79, p.1-7.

Pyster, Arthur B. (1980) "Compiler Design and Construction ", Van Nostrand Reinhold, N. Y.

Robinson, k. (1980) "The Design of a Successor to Pascal", Language Design and Programming Methodology , Springer-Verlag, Berlin.

Sakoda, James M. (1979) "Dystal2: A General Purpose Extension of Fortran", Sigplan Notices 14,1 Jan/79, p.77-90.

Six, H.-W. (1980) "A Framework for Data Structures ", CS Tech. Report No. 80-CS-26, McMaster University.

Skordalakis, E., G. Papakonstantinou (1978) "Coroutines in Fortran", Sigplan Notices 13,9 Sept/78, p.76-84.

Solntseff, N., A. Yezerski (1974) "A Survey of Extensible Languages", Annual Review in Automatic Programming 7 , Pergamon Press, Oxford.

Steele, C. A., A. E. Sedgwick (1974) "Deft - A Disciplined Extension of Fortran ", Technical Report No. 62, University of Toronto.

Wagener, J. L. (1978) "Tutorial: Common Features in Looping Proposals", Minutes of 63rd Meeting , X3J3/104, ANSI, p.37-45.

Wallis, Peter J. L. (1980) "External Representations of Objects of User-Defined Types", ACM Transactions on Programming Languages and Systems 2,2 Apr/80, p.137 152.

Wirth, N. (1977) "Modula: A Language for Modular Multiprogramming", Software Practise and Experience , Jan/77, p.3-35.

Wirth, N. (1980) "The Module: A System Structuring Facility in High-Level Programming Languages", Language Design and Programming Methodology , Springer-Verlag, Berlin.

Wulf, W. A., R. L. London, M Shaw (1976) "An Introduction to the construction and verification of Alphard Programs", IEEE Trans. Softw. Eng. , Dec/76, p.253-264.

Wulf, William A., Mary Shaw (1977) "Abstraction and Verification in Alphard - Defining and Specifying Iterators and Generators", CACM 20 Aug/77 p.553-564.

Appendix A

This appendix contains a complete Mytran translation of several FORMs. The program at the end contains declarations of several of the types that may be constructed from these FORMs.

```
FORM INTFORM INTEGER
  EXPORT
    INTCMP AS COMPARE
    INTASGN AS ASSIGN
  FUNCTION INTCMP (I1,I2)
    INTCMP : INTEGER
  BEGIN
    IF
      <I1 LT I2> INTCMP = -1
      <I1 EQ I2> INTCMP = 0
      <I1 GT I2> INTCMP = 1
    FI
  END
  SUBROUTINE INTASGN (I1,I2)
  BEGIN I2 = I1 END
ENDFORM

FORM STRING (ENTRY:FORM, LENGTH:INTEGER)
  ARRAY (LENGTH) OF ENTRY
  IMPORT
    FROM ENTRY
           COMPARE (S1,S2) : INTEGER
           ASSIGN (S1,S2)
  EXPORT
    STRCMP AS COMPARE
    STRCPY AS ASSIGN

  FUNCTION STRCMP (S1,S2)
    S1,S2 : STRING
    STRCMP : INTEGER
  BEGIN
    LOOP
      GIVEN I = 1
      WHILE
        <I LE LENGTH>
        <S1 (I).COMPARE (S1 (I), S2 (I)) EQ 0>
      LOOPBY I = I + 1
    ENDLOOP
    IF
      <I GT LENGTH> STRCMP = 0
      <I LE LENGTH> STRCMP = S1 (I).COMPARE (S1 (I), S2 (I))
    FI
  END
```

```
  SUBROUTINE STRCPY (S1,S2)
    S1,S2 : STRING
  BEGIN
    LOOP
      GIVEN I = 1
      WHILE <I LE LENGTH>
      DO CALL S1 (I).ASSIGN (S1 (I), S2 (I))
      LOOPBY I = I + 1
    ENDLOOP
  END
ENDFORM

FORM TREE (KEYTYPE:FORM, VALUETYPE:FORM, MAX:INTEGER)
  RECORD
    ROOT, FREENODE : INTEGER
    TRUNK : ARRAY (MAX) OF RECORD
                             KEY : KEYTYPE
                             VALUE : VALUETYPE
                             LEFT,RIGHT : INTEGER
                           ENDREC
  ENDREC

  IMPORT
    FROM KEYTYPE
           COMPARE (K1,K2) : INTEGER
           ASSIGN (K1,K2)
    FROM VALUETYPE
           ASSIGN (V1,V2)

  EXPORT
    TRSRCH AS SEARCH
    TRADD AS ADD
    TRINIT AS INIT

  SUBROUTINE TRINIT (T)
    T : TREE
  BEGIN T.ROOT = 0; T.FREENODE = 1 END

  SUBROUTINE TRADD (T,K,V,OK)
    K : KEYTYPE
    V : VALUETYPE
    T : TREE
    OK, FOUND : LOGICAL
    ADDRESS : INTEGER
  BEGIN
    CALL T.TRSRCH (T,K,FOUND,ADDRESS)
    IF
      <FOUND> OK = FALSE
      <NOT FOUND>
```

```
        IF
          <T.FREENODE GE MAX> OK = FALSE
          <T.FREENODE LT MAX>
            N = T.FREENODE; T.FREENODE = T.FREENODE + 1
            CALL K.ASSIGN (K,T.TRUNK (N).KEY)
            CALL V.ASSIGN (V,T.TRUNK (N).VALUE)
            T.TRUNK (N).LEFT = Ø
            T.TRUNK (N).RIGHT = Ø
            IF
              <ADDRESS EQ Ø> T.ROOT = N
              <ADDRESS NE Ø>
                D = K.COMPARE (K,T.TRUNK (ADDRESS).KEY)
                IF
                  <D EQ -1> T.TRUNK (ADDRESS).LEFT = N
                  <D EQ 1> T.TRUNK (ADDRESS).RIGHT = N
                FI
            FI
        FI
    FI
END

  SUBROUTINE TRSRCH (T,K,FOUND,ADDRESS)
    T : TREE
    K : KEYTYPE
    FOUND : LOGICAL
    ADDRESS : INTEGER
  BEGIN
    LOOP
      GIVEN
        ADDRESS = Ø
        NEXT = T.ROOT
      WHILE
        <NEXT NE Ø>
        D = K.COMPARE (K,T.TRUNK (NEXT).KEY)
        <D NE Ø>
      LOOPBY
        ADDRESS = NEXT
        IF
          <D EQ -1> NEXT = T.TRUNK (NEXT).LEFT
          <D EQ 1> NEXT = T.TRUNK (NEXT).RIGHT
        FI
    ENDLOOP
    IF
      <NEXT EQ Ø> FOUND = FALSE
      <NEXT NE Ø> FOUND = TRUE
    FI
  END

ENDFORM
```

```
FORM LOGFORM LOGICAL ENDFORM

PROGRAM TEST
C
C TII is a tree with integer keys and integer values
C
  TII : TREE (INTFORM,INTFORM,20)
C
C TIS is a tree with integer keys an strings of 5 integers
C for values.
C
  TIS : TREE (INTFORM,STRING (INTFORM,5),20)
C
C TSI is a tree with strings of 3 integers for keys and
C integer values.
C
  TSI : TREE (STRING (INTFORM,3),INTFORM,10)
C
C TSS is a tree with strings of 3 integers for keys and
C strings of 7 integers for values.
C
  TSS : TREE (STRING (INTFORM,3),STRING (INTFORM,7),30)
  I,J : INTFORM
  S1 : STRING (INTFORM,3)
  S2 : STRING (INTFORM,5)
  S3 : STRING (INTFORM,7)
  OK : LOGICAL
BEGIN
C
C This program simply shows the nature of the references that
C may be made to these variables and the translation of those
C references.
C
  TII.TRUNK (I).KEY = 1
  TSI.TRUNK (I).KEY (1) = 1
  TSS.TRUNK (I).VALUE (J) = 1
  CALL TII.TRADD (TII,I,J,OK)
  CALL TSS.TRADD (TSS,S1,S3,OK)
END
```

```
      FUNCTIONINTCMP(TYPE,I1,I2)
      INTEGERINTCMP
      INTEGERNAME(6),TYPE(1)
      DATA NAME/"I","N","T","C","M","P"/
      LABEL=0
      IF(.NOT.(I1.LT.I2))GOTO 20
      CALL TEST01(LABEL,LINE, 1,   9,NAME)
      GOTO   20
30    CONTINUE
      INTCMP=-1
      GOTO  10
20    CONTINUE
      IF(.NOT.(I1.EQ.I2))GOTO 40
      CALL TEST01(LABEL,LINE, 2,  10,NAME)
      GOTO   40
50    CONTINUE
      INTCMP=0
      GOTO  10
40    CONTINUE
      IF(.NOT.(I1.GT.I2))GOTO 60
      CALL TEST01(LABEL,LINE, 3,  11,NAME)
      GOTO   60
70    CONTINUE
      INTCMP=1
      GOTO  10
60    CONTINUE
      IF(LABEL.EQ.0)CALL ERR001(   8,NAME)
      GOTO(30,50,70),LABEL
10    CONTINUE
      RETURN
      END

      SUBROUTINEINTASGN(TYPE,I1,I2)
      INTEGERNAME(7),TYPE(1)
      DATA NAME/"I","N","T","A","S","G","N"/
      I2=I1
      RETURN
      END

      FUNCTIONSTRCMP(TYPE,S1,S2)
      INTEGERS1(1),S2(1)
      INTEGERSTRCMP
      INTEGERCOMPARE
      INTEGERNAME(6),TYPE(1)
      DATA NAME/"S","T","R","C","M","P"/
      I=1
80    CONTINUE
      IF(.NOT.(I.LE.TYPE(3)))GOTO 90
      IF(.NOT.(COMPARE(TYPE(TYPE(2)),S1(DOPE1(LOCAL(4),I)*TYP
```

```
   XE(6)+1),S2(DOPE1(LOCAL(4),I)*TYPE(6)+1)).EQ.0))GOTO 90
    I=I+1
    GOTO  80
 90 CONTINUE
    LABEL=0
    IF(.NOT.(I.GT.TYPE(3)))GOTO 110
    CALL TEST01(LABEL,LINE, 1,  39,NAME)
    GOTO  110
120 CONTINUE
    STRCMP=0
    GOTO 100
110 CONTINUE
    IF(.NOT.(I.LE.TYPE(3)))GOTO 130
    CALL TEST01(LABEL,LINE, 2,  40,NAME)
    GOTO  130
140 CONTINUE
    STRCMP=COMPARE(TYPE(TYPE(2)),S1(DOPE1(LOCAL(4),I)*TYPE(
   X6)+1),S2(DOPE1(LOCAL(4),I)*TYPE(6)+1))
    GOTO 100
130 CONTINUE
    IF(LABEL.EQ.0)CALL ERR001(  38,NAME)
    GOTO(120,140),LABEL
100 CONTINUE
    RETURN
    END

    SUBROUTINESTRCPY(TYPE,S1,S2)
    INTEGERS1(1),S2(1)
    INTEGERCOMPARE
    INTEGERNAME(6),TYPE(1)
    DATA NAME/"S","T","R","C","P","Y"/
    I=1
150 CONTINUE
    IF(.NOT.(I.LE.TYPE(3)))GOTO 160
    CALLASSIGN(TYPE(TYPE(2)),S1(DOPE1(LOCAL(4),I)*TYPE(6)+1
   X),S2(DOPE1(LOCAL(4),I)*TYPE(6)+1))
    I=I+1
    GOTO 150
160 CONTINUE
    RETURN
    END
```

```
      SUBROUTINETRINIT(TYPE,T)
      INTEGERT(1)
      INTEGERCOMPARE
      INTEGERNAME(6),TYPE(1)
      DATA NAME/"T","R","I","N","I","T"/
      T(0+1)=0
      T(1+1)=1
      RETURN
      END

      SUBROUTINETRADD(TYPE,T,K,V,OK)
      INTEGERK(1)
      INTEGERV(1)
      INTEGERT(1)
      LOGICALOK,FOUND
      INTEGERADDRESS
      INTEGERCOMPARE
      INTEGERNAME(5),TYPE(1)
      DATA NAME/"T","R","A","D","D"/
      CALLTRSRCH(TYPE,T(1),K(1),FOUND,ADDRESS)
      LABEL=0
      IF(.NOT.(FOUND))GOTO 180
      CALL TEST01(LABEL,LINE, 1,  90,NAME)
      GOTO  180
190   CONTINUE
      OK=.FALSE.
      GOTO 170
180   CONTINUE
      IF(.NOT.(.NOT.FOUND))GOTO 200
      CALL TEST01(LABEL,LINE, 2,  91,NAME)
      GOTO  200
210   CONTINUE
      LABEL=0
      IF(.NOT.(T(1+1).GE.TYPE(4)))GOTO 230
      CALL TEST01(LABEL,LINE, 1,  93,NAME)
      GOTO  230
240   CONTINUE
      OK=.FALSE.
      GOTO 220
230   CONTINUE
      IF(.NOT.(T(1+1).LT.TYPE(4)))GOTO 250
      CALL TEST01(LABEL,LINE, 2,  94,NAME)
      GOTO  250
260   CONTINUE
      N=T(1+1)
      T(1+1)=T(1+1)+1
      CALLASSIGN(TYPE(TYPE(2)),K(1),T(2+DOPE1(TYPE(5),N)*TYPE
     X(10)+0+1))
      CALLASSIGN(TYPE(TYPE(3)),V(1),T(2+DOPE1(TYPE(5),N)*TYPE
```

```
   X(10)+TYPE(7)+1))
    T(2+DOPE1(TYPE(5),N)*TYPE(10)+TYPE(8)+1)=0
    T(2+DOPE1(TYPE(5),N)*TYPE(10)+TYPE(9)+1)=0
    LABEL=0
    IF(.NOT.(ADDRESS.EQ.0))GOTO 280
    CALL TEST01(LABEL,LINE, 1, 101,NAME)
    GOTO  280
290 CONTINUE
    T(0+1)=N
    GOTO 270
280 CONTINUE
    IF(.NOT.(ADDRESS.NE.0))GOTO 300
    CALL TEST01(LABEL,LINE, 2, 102,NAME)
    GOTO  300
310 CONTINUE
    D=COMPARE(TYPE(TYPE(2)),K(1),T(2+DOPE1(TYPE(5),ADDRESS)
   X*TYPE(10)+0+1))
    LABEL=0
    IF(.NOT.(D.EQ.-1))GOTO 330
    CALL TEST01(LABEL,LINE, 1, 105,NAME)
    GOTO  330
340 CONTINUE
    T(2+DOPE1(TYPE(5),ADDRESS)*TYPE(10)+TYPE(8)+1)=N
    GOTO 320
330 CONTINUE
    IF(.NOT.(D.EQ.1))GOTO 350
    CALL TEST01(LABEL,LINE, 2, 106,NAME)
    GOTO  350
360 CONTINUE
    T(2+DOPE1(TYPE(5),ADDRESS)*TYPE(10)+TYPE(9)+1)=N
    GOTO 320
350 CONTINUE
    IF(LABEL.EQ.0)CALL ERR001( 104,NAME)
    GOTO(340,360),LABEL
320 CONTINUE
    GOTO 270
300 CONTINUE
    IF(LABEL.EQ.0)CALL ERR001( 100,NAME)
    GOTO(290,310),LABEL
270 CONTINUE
    GOTO 220
250 CONTINUE
    IF(LABEL.EQ.0)CALL ERR001(  92,NAME)
    GOTO(240,260),LABEL
220 CONTINUE
    GOTO 170
200 CONTINUE
    IF(LABEL.EQ.0)CALL ERR001(  89,NAME)
    GOTO(190,210),LABEL
```

```
170 CONTINUE
    RETURN
    END

    SUBROUTINETRSRCH(TYPE,T,K,FOUND,ADDRESS)
    INTEGERT(1)
    INTEGERK(1)
    LOGICALFOUND
    INTEGERADDRESS
    INTEGERCOMPARE
    INTEGERNAME(6),TYPE(1)
    DATA NAME/"T","R","S","R","C","H"/
    ADDRESS=0
    NEXT=T(0+1)
370 CONTINUE
    IF(.NOT.(NEXT.NE.0))GOTO 380
    D=COMPARE(TYPE(TYPE(2)),K(1),T(2+DOPE1(TYPE(5),NEXT)*TY
   XPE(10)+0+1))
    IF(.NOT.(D.NE.0))GOTO 380
    ADDRESS=NEXT
    LABEL=0
    IF(.NOT.(D.EQ.-1))GOTO 400
    CALL TEST01(LABEL,LINE, 1, 130,NAME)
    GOTO  400
410 CONTINUE
    NEXT=T(2+DOPE1(TYPE(5),NEXT)*TYPE(10)+TYPE(8)+1)
    GOTO 390
400 CONTINUE
    IF(.NOT.(D.EQ.1))GOTO 420
    CALL TEST01(LABEL,LINE, 2, 131,NAME)
    GOTO  420
430 CONTINUE
    NEXT=T(2+DOPE1(TYPE(5),NEXT)*TYPE(10)+TYPE(9)+1)
    GOTO 390
420 CONTINUE
    IF(LABEL.EQ.0)CALL ERR001( 129,NAME)
    GOTO(410,430),LABEL
390 CONTINUE
    GOTO 370
380 CONTINUE
    LABEL=0
    IF(.NOT.(NEXT.EQ.0))GOTO 450
    CALL TEST01(LABEL,LINE, 1, 135,NAME)
    GOTO  450
460 CONTINUE
    FOUND=.FALSE.
    GOTO 440
450 CONTINUE
    IF(.NOT.(NEXT.NE.0))GOTO 470
```

```
      CALL TEST01(LABEL,LINE, 2, 136,NAME)
      GOTO  470
480 CONTINUE
      FOUND=.TRUE.
      GOTO 440
470 CONTINUE
      IF(LABEL.EQ.0)CALL ERR001( 134,NAME)
      GOTO(460,480),LABEL
440 CONTINUE
      RETURN
      END

      INTEGER FUNCTION COMPARE(TYPE,S1,S2)
      INTEGER TYPE (1)
      INTEGERINTCMP,STRCMP
      GOTO(1,2),TYPE(1)
    1 CONTINUE
      COMPARE=INTCMP(TYPE,S1,S2)
      GOTO 999
    2 CONTINUE
      COMPARE=STRCMP(TYPE,S1,S2)
      GOTO 999
999 RETURN
      END

      SUBROUTINE ASSIGN(TYPE,S1,S2)
      INTEGER TYPE (1)
      GOTO(1,2),TYPE(1)
    1 CONTINUE
      CALL INTASGN(TYPE,S1,S2)
      GOTO 999
    2 CONTINUE
      CALL STRCPY(TYPE,S1,S2)
      GOTO 999
999 RETURN
      END
```

```
 PROGRAMTEST
 INTEGERTII(82)
 INTEGERTIS(162)
 INTEGERTSI(62)
 INTEGERTSS(362)
 INTEGERI(1),J(1)
 INTEGERS1(3)
 INTEGERS2(5)
 INTEGERS3(7)
 LOGICALOK
 INTEGERLOCAL(94),NAME(4)
 DATA LOCAL/3,11,12,20,1,20,1,2,3,4,1,1,3,11,12,20,1,20,
X1,6,7,8,1,2,7,5,1,5,1,1,3,11,18,10,1,10,3,4,5,6,2,7,3,1
X,3,1,1,1,3,11,18,30,1,30,3,10,11,12,2,7,3,1,3,1,1,2,7,7
X,1,7,1,1,1,2,7,3,1,3,1,1,2,7,5,1,5,1,1,2,7,7,1,7,1,1/
 DATA NAME/"T","E","S","T"/
 TII(2+DOPE1(LOCAL(5),I(1))*4+0+1)=1
 TSI(2+DOPE1(LOCAL(35),I(1))*6+0+DOPE1(LOCAL(4),1)*TYPE(
X6)+1)=1
 TSS(2+DOPE1(LOCAL(53),I(1))*12+3+DOPE1(LOCAL(4),J(1))*T
XYPE(6)+1)=1
 CALLTRADD(LOCAL(1),TII(1),I(1),J(1),OK)
 CALLTRADD(LOCAL(49),TSS(1),S1(1),S3(1),OK)
 RETURN
 END
```

Appendix B

The Mytran symbol table is stored in a linked structure with variable length entries and links in two directions, called "down" and "across". Different entries contain different types of information. The following entries are used to store FORM descriptions.

FormNameEntry contains 1) Name of FORM, 2) number of words in dope vector, 3) FORM number. It is joined across to the next FormNameEntry and down to the HeaderEntry.

HeaderEntry contains 1) number of parameters, and is joined across to the DummyParmEntries and down to the StructureEntry.

DummyParmEntry contains 1) the parameter name, 2) the parameter type (FORM or INTEGER) and is joined across to the next DummyParmEntry.

StructureEntry contains 1) the type of structure (Record, Array, Fortran, Form, Parm or Current), 2) the size of the structure if it is known, 3) some information depending on the type of structure. This is the number of fields in a record, the number of ranges in an array, the type of a

fortran scalar, the number of a parm or the pointer to the current form. The type of entry to which it is joined is also determined by the type of structure. A record entry is joined across to its FieldEntries. An array entry is joined across to its base StructureEntry, and down to its RangeEntries. A form entry is joined down to its FormSpecEntry. A parm entry is joined across to its parm name.

FieldEntry contains 1) the field name, 2) the offset to the field if it is known. It is joined across to the next FieldEntry and down to its StructureEntry.

RangeEntry contains 1) the type of entry which may be a value or a parameter, 2) the value or parameter number. It is joined down to the next RangeEntry.

FormSpecEntry contains 1) the name of the FORM, 2) the starting address in the dope vector, 3) a pointer to the FORM. It is joined across to its ActualParmEntries.

ActualParmEntry contains 1) the type of entry (FORM or INTEGER) and is joined across to the next ActualParmEntry and down to its value or FormSpecEntry.

ImportEntry contains 1) the number of FORM parameters which import something. It is joined across to its ImportFormEntries and down to the ExportEntry.

ImportFormEntry contains 1) the FORM parameter name, 2) the number of imported items, 3) the parameter number. It is joined across to the next ImportFormEntry and down to its ImportItemEntries.

ImportItemEntry contains 1) the type of imported item (Subroutine, or Function) and is joined across to its ImportSubEntry and down to the next ImportItemEntry.

ImportSubEntry contains 1) the subroutine or function name, 2) the number of parameters and if it is a function, is joined down to its type.

ExportEntry contains 1) the number of exported items and is joined across to its ExportItemEntries.

ExportItemEntry contains 1) its local name, 2) its alias name and is joined across to the next ExportItemEntry.

The following entries are used to store variable declarations and their expanded type descriptions.

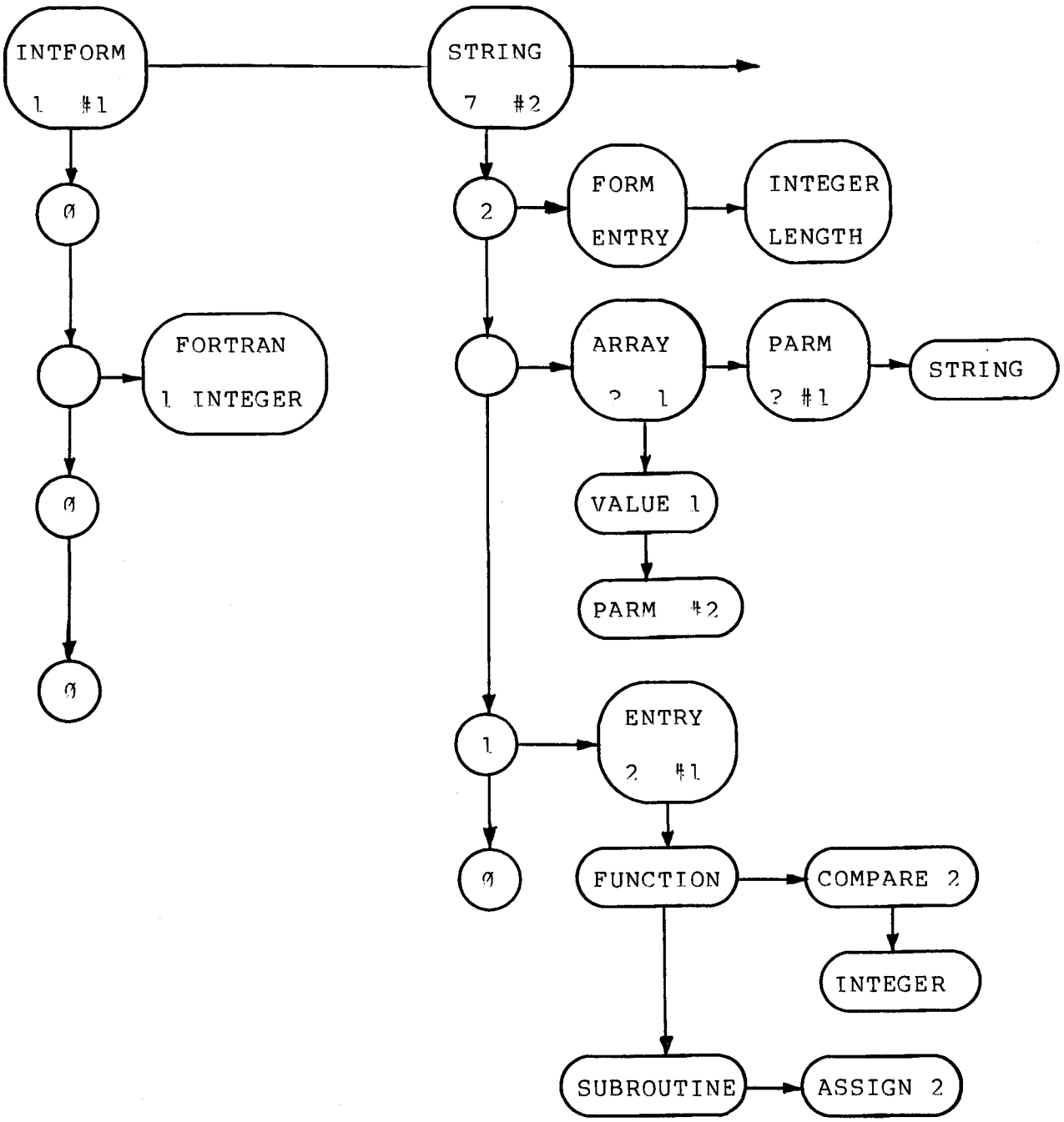
VarNameEntry contains 1) the variable name, 2) the address of the start of its dope vector. It is joined across to the next VarNameEntry and down to its StructureEntry.

StructureEntry is the same as the StructureEntry for a FORM except that if the structure is the current form, it is joined down to an expanded StructureEntry for the current form.

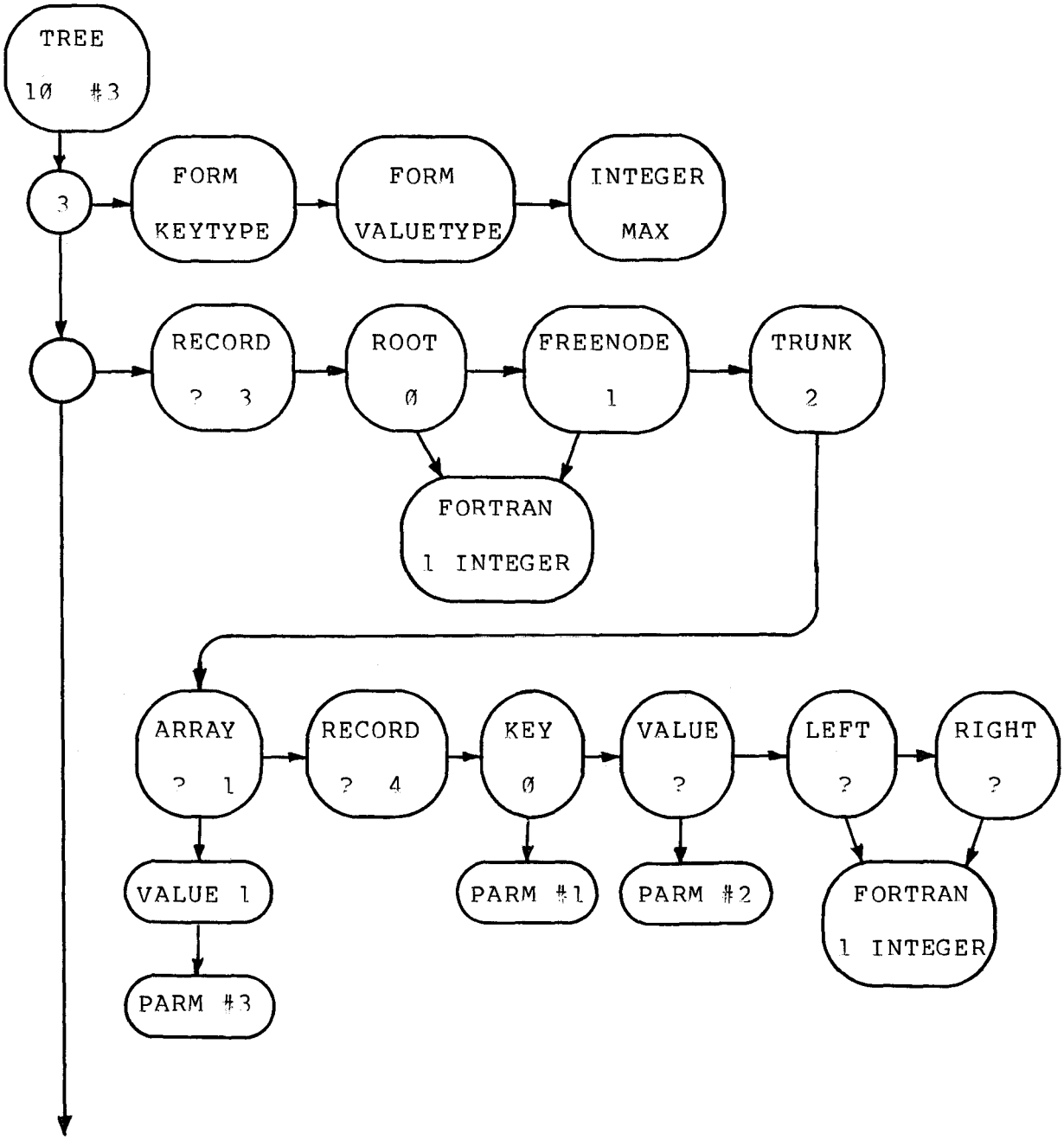
The FieldEntry, FormSpecEntry and ActualParmEntry are all the same as for a FORM except that where sizes and offsets are stored, they are now either known values or addresses in the dope vector.

RangeEntry contains 1) the starting address in the dope vector of the range list.

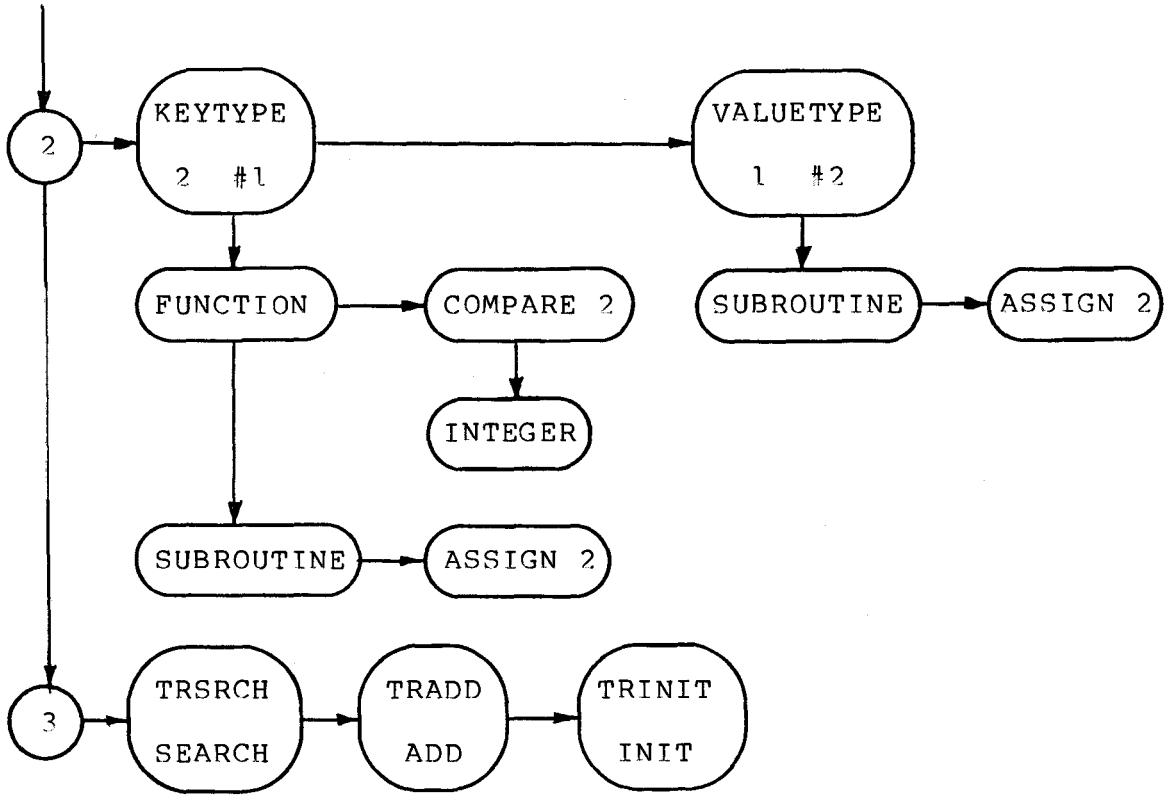
The rest of this appendix contains some graphic representations of the symbol tables for some of the FORMs and types described in the example in Appendix A.



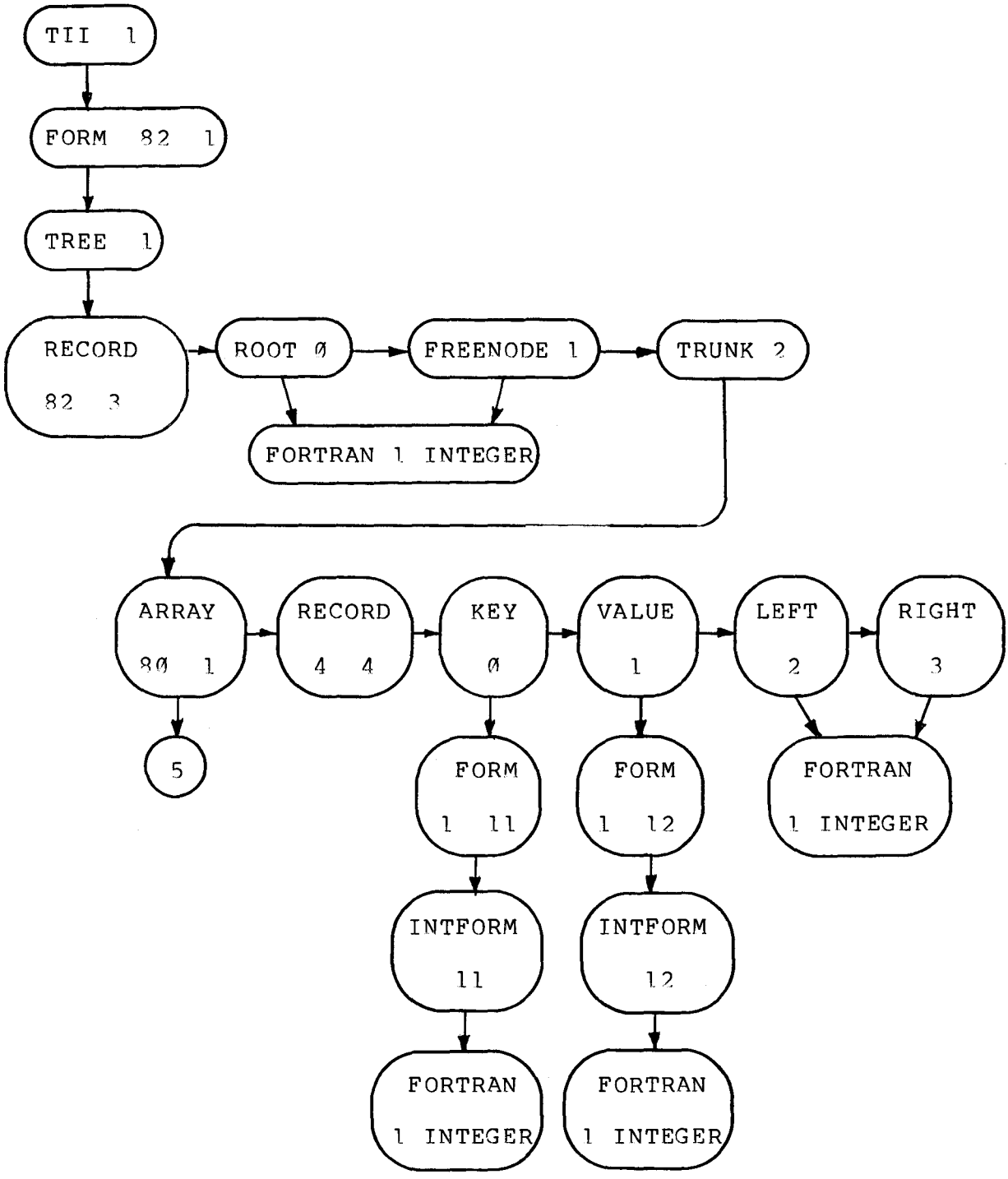
Symbol table for INTFORM and STRING FORMs.



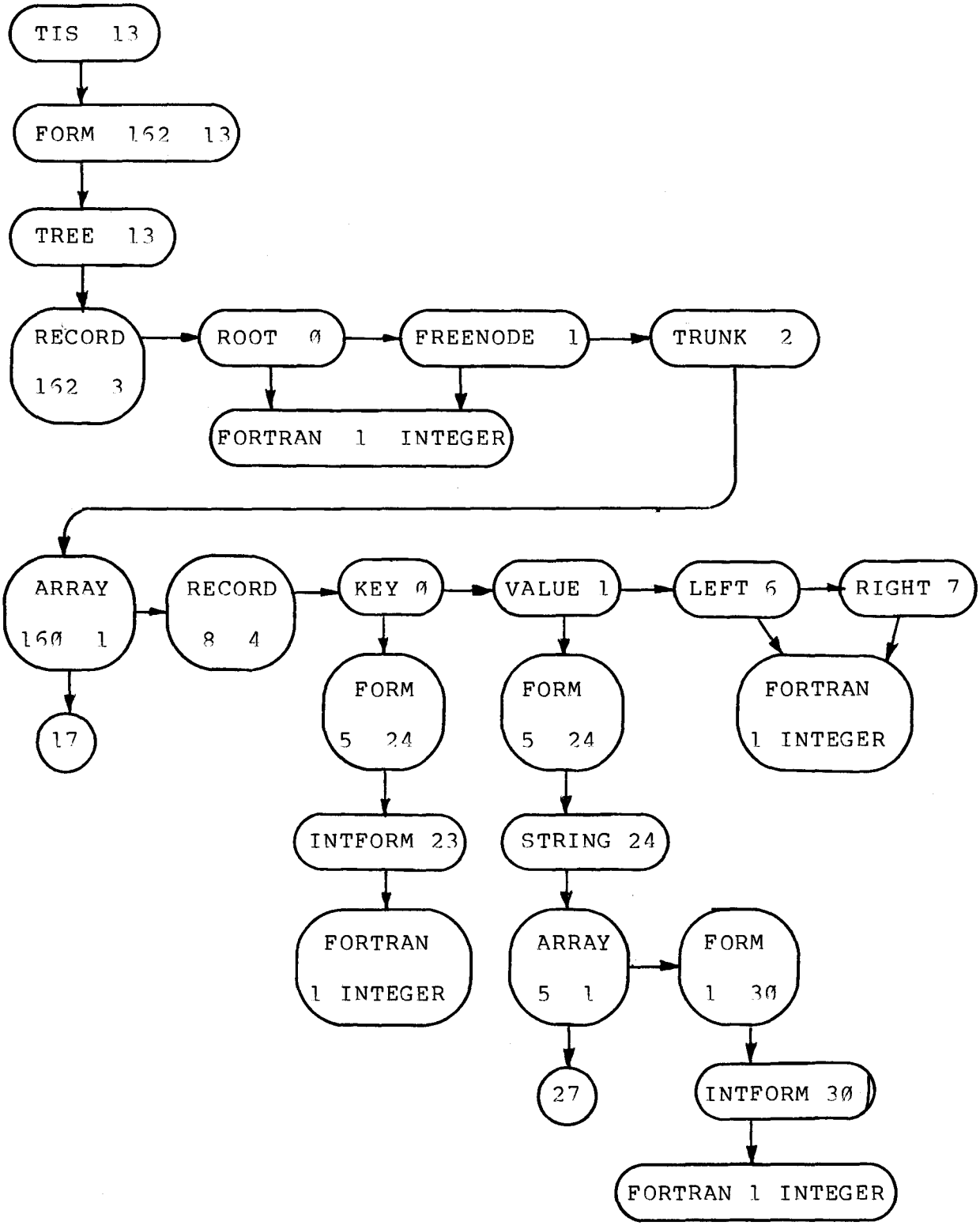
Symbol table for TREE FORM, part i.



Symbol table for TREE FORM, part ii.



Symbol table for type of TII.

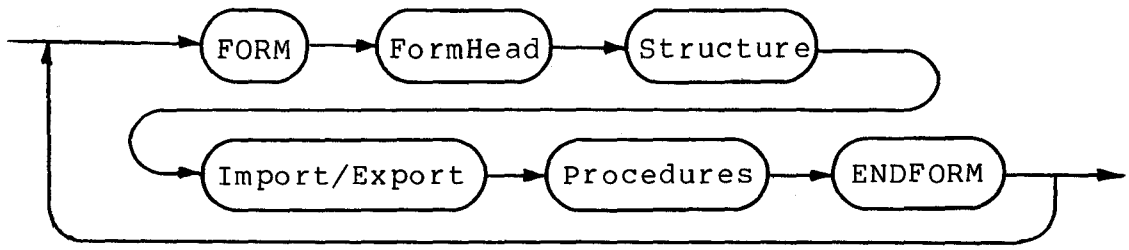


Symbol table for type of TIS.

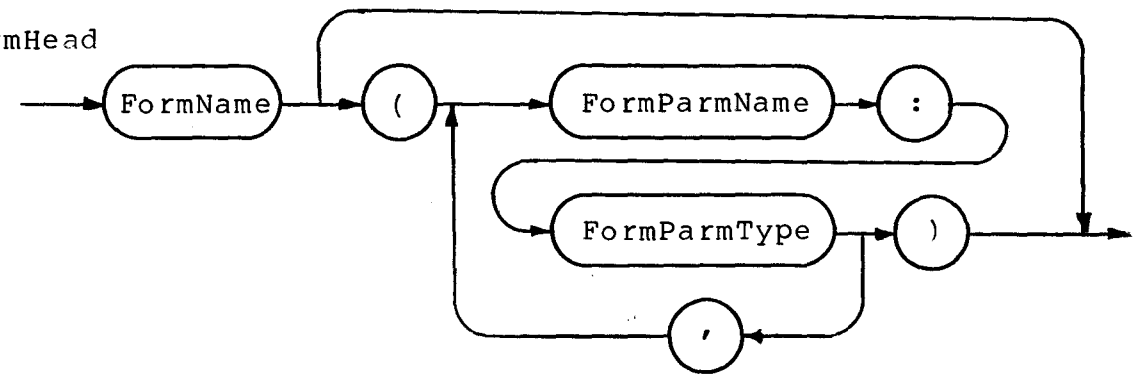
Appendix C

This appendix defines the syntax of Mytran. Any non-terminal ending in "Name" is equivalent to the non-terminal Name which is a letter followed by any number of letters or digits. The non-terminals Lower, Upper and Constant are equivalent to IntegerConstant which is a digit followed by any number of digits.

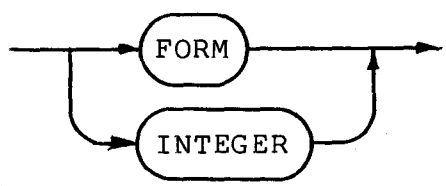
Forms



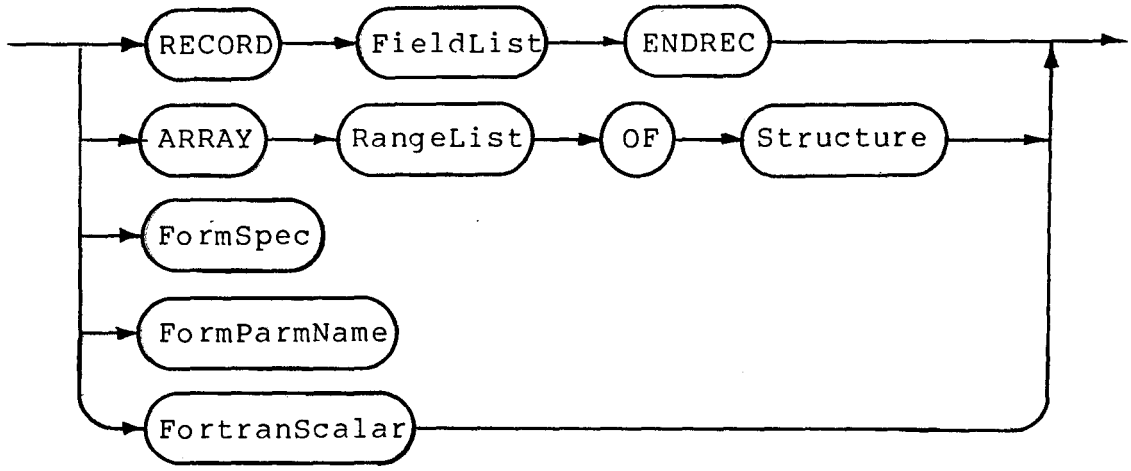
FormHead



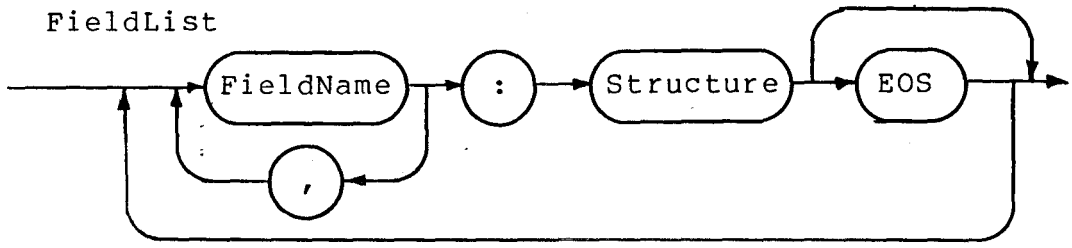
FormParmType



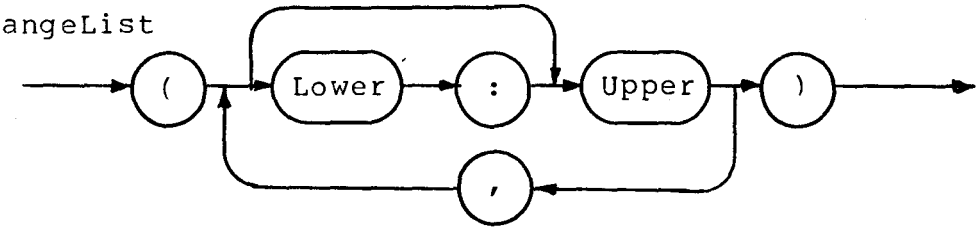
Structure



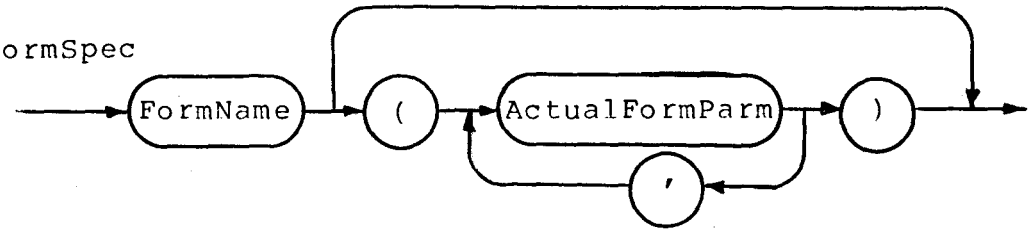
FieldList



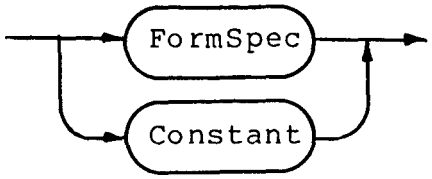
RangeList



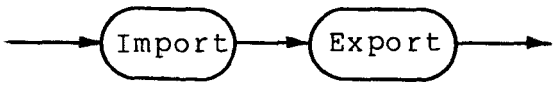
FormSpec



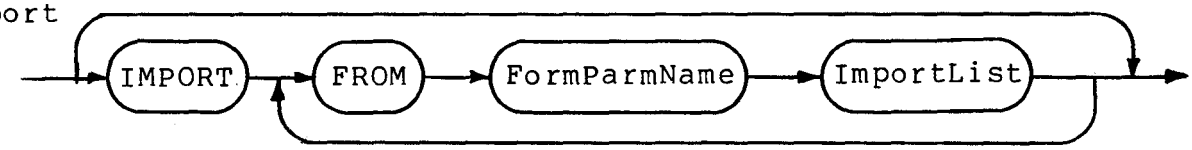
ActualFormParm



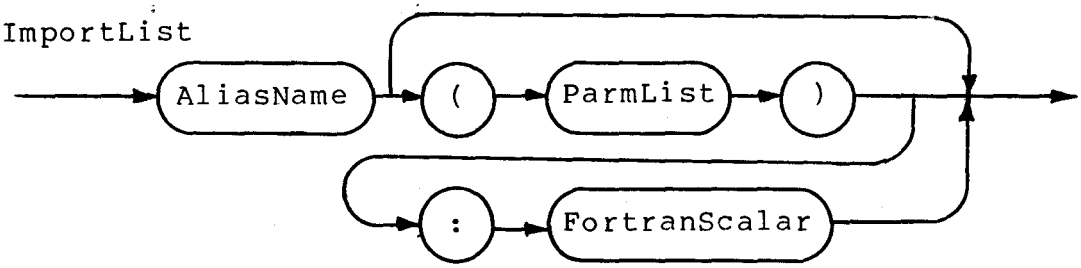
Import/Export



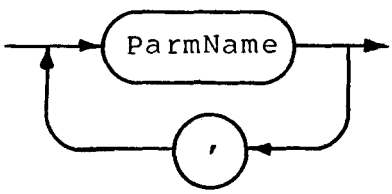
Import



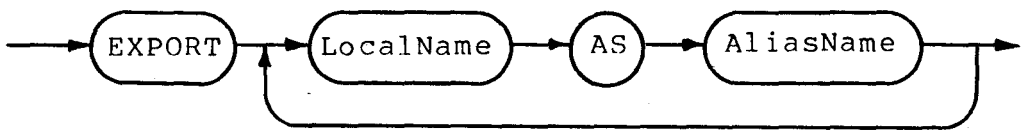
ImportList



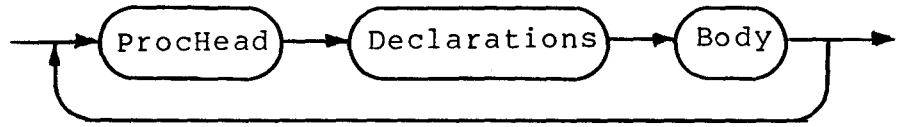
ParmList



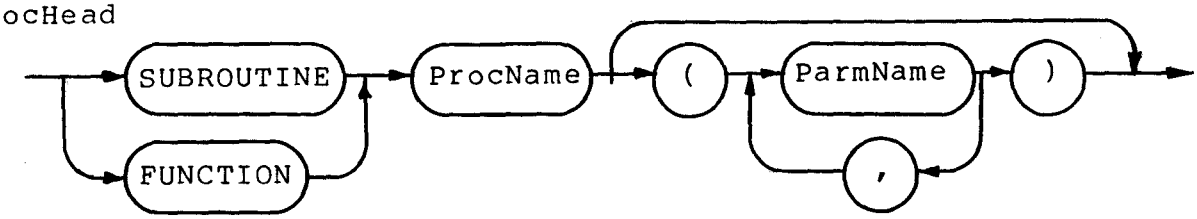
Export



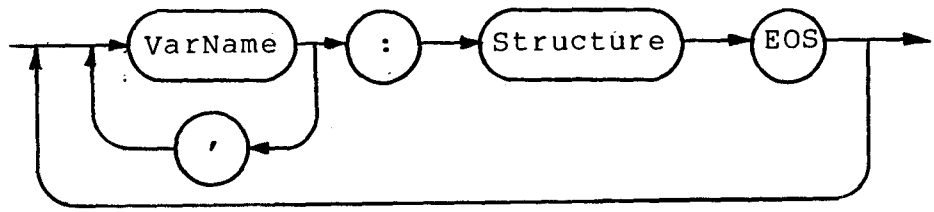
Procedures



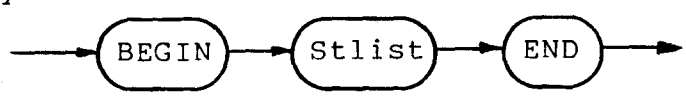
ProcHead



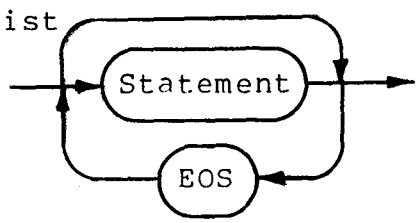
Declarations



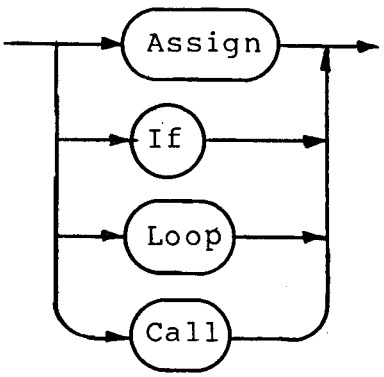
Body



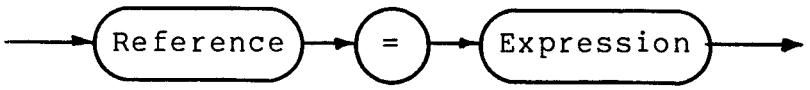
StList



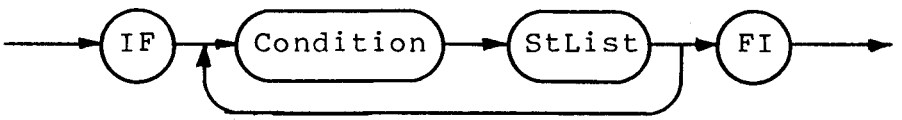
Statement



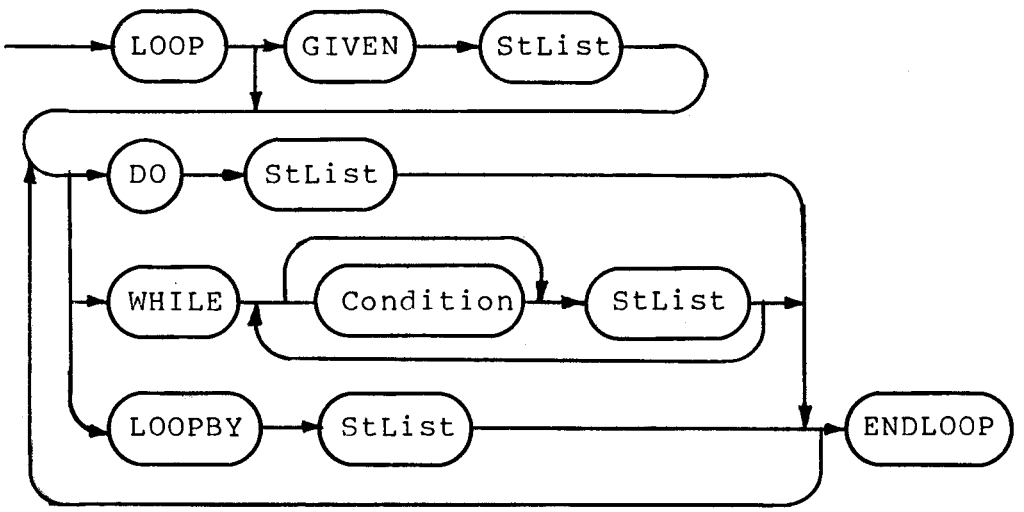
Assign



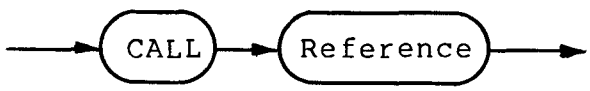
If



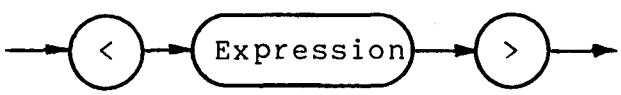
Loop



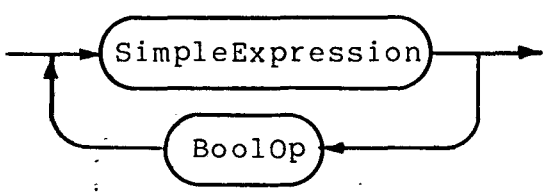
Call



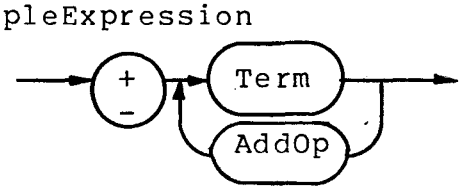
Condition



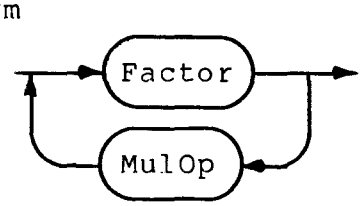
Expression



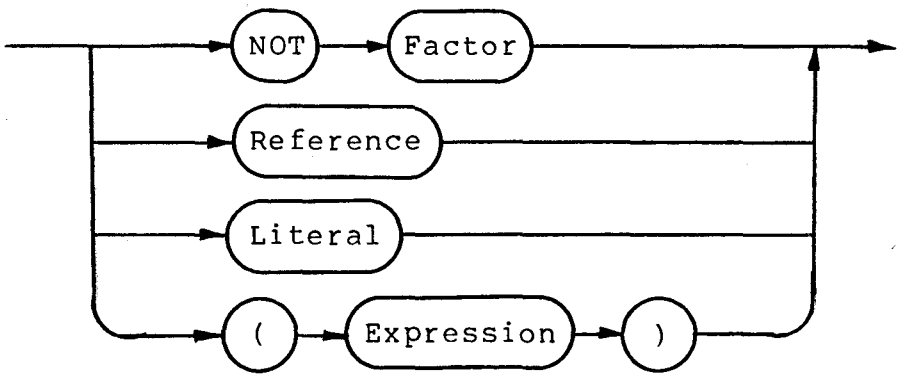
SimpleExpression



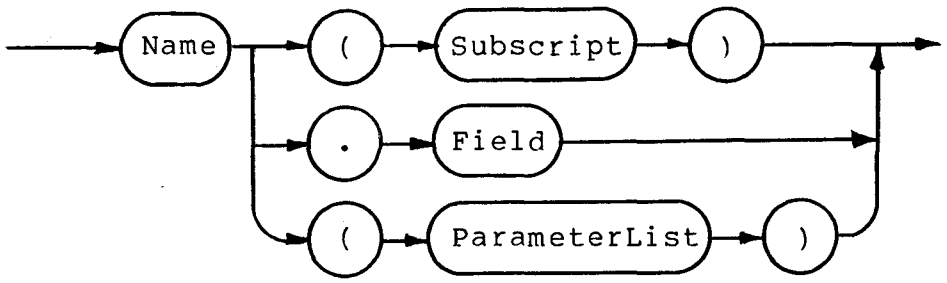
Term



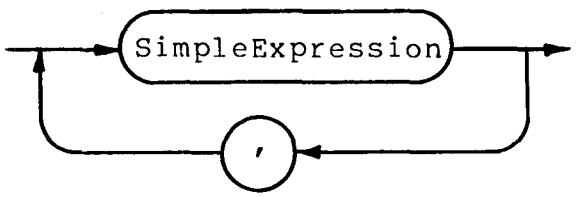
Factor



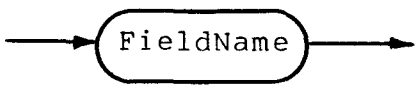
Reference



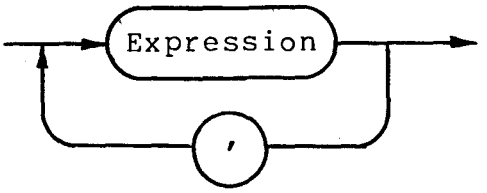
Subscript



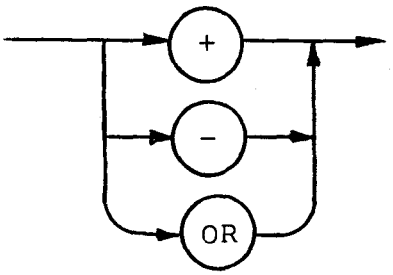
Field



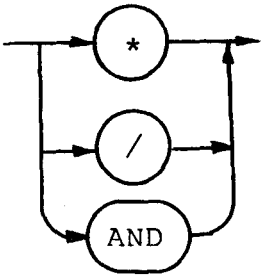
ParameterList



AddOp



MulOp



Appendix D

This appendix contains some of the code for the Mytran preprocessor. The routines which parse and generate the symbol table for FORMs, the routines which expand a form specification into a type, and the routines which generate access expressions from Mytran variable references, are included. The routines which parse and generate code for statements and expressions and all of the service routines are not included.

```
      SUBROUTINE FORM (OK)
      IMPLICIT INTEGER (A-Z)
      LOGICAL OK
      COMMON /T/ TOKEN
      COMMON /SYMTAB/ FRMTAB, FRMLST, VARTAB, VARLST
      COMMON /CURFRM/ FRMPTR, HDPTR, STRPTR, IMPPTR, EXPPTR
      COMMON /SCAN/ FORMM, DECL
      COMMON /FNUM/ FNUM
      LOGICAL FORMM, DECL
      INTEGER T (3)
C
C Set LOCAL address to Ø and allocate a space for the
C Form Number.
C
      WRITE (3,*) "Enter FORM"
      FORMM = .TRUE.; DECL = .FALSE.
      WRITE (6,*) " In FORM"
      CALL GETRACE
      IF (TOKEN .NE. @FORM) OK = .FALSE.
         (TOKEN .EQ. @FORM)
           CALL SETLOC
           ADDR = NEWLOC (1)
C
C Scan the header, structure, import/export lists and
C procedures of a form and join them together in the symbol table.
C
            CALL FORMHD (FRMPTR,HDPTR,OK)
            IF (.NOT. OK)
               (OK)
                 CALL JOINLL (FRMLST, FRMPTR, @ACROSS)
                 FRMLST = FRMPTR
                 CALL NEWLL (DUMPTR,1)
                 CALL JOINLL (HDPTR, DUMPTR, @DOWN)
                 CALL STRUCT (STRPTR, OK)
                 CALL JOINLL (DUMPTR, STRPTR, @ACROSS)
C
C Find out how many words in LOCAL were used by the form,
C and store it in the table.
C
                 CALL GETLL (FRMPTR,T)
                 T (2) = CURLOC (X); T (3) = FNUM
                 CALL STORLL (FRMPTR,T)
                 IF (.NOT. OK)
                     (OK) CALL IMPEXP (IMPPTR, OK)
                       IF (.NOT. OK)
                           (OK)
                             CALL JOINLL (DUMPTR, IMPPTR, @DOWN)
                             CALL SNAPLL
                             CALL SNAPAS
```

```
                              CALL PROC (OK)
                              CALL RSETLL; CALL RSETAS
                              VARLST = VARTAB
                              IF (.NOT. OK)
                                  (OK)
                                    IF (TOKEN .EQ. @ENDFORM) CALL GETTKN
                                       (TOKEN .NE. @ENDFORM) OK = .FALSE.
                                    FI
                              FI
                         FI
                    FI
               FI
          FI
          WRITE (3,*) "Leave FORM"
          RETURN
          END

          SUBROUTINE FORMHD (FRMPTR,HDPTR,OK)
          IMPLICIT INTEGER (A-Z)
          COMMON /T/ TOKEN
          COMMON /TS/ TKNSTR (22)
          INTEGER T (2)
          LOGICAL OK
C
C A form header is a name followed by Ø or more parameters.
C Store the name in the form entry.  The form entry points down
C to the header enter.  The header entry contains the number of
C parameters and points across to the list of parameter entries.
C
          WRITE (3,*) "Enter FORMHD"
          CALL GETTKN
          IF (TOKEN .NE. @NAME) OK = .FALSE.
             (TOKEN .EQ. @NAME)
               CALL NEWLL (FRMPTR, 3)
               CALL ADDAS (FRMNAM, TKNSTR)
               T (1) = FRMNAM
               CALL STORLL (FRMPTR, T)
               CALL NEWLL (HDPTR, 1)
               CALL JOINLL (FRMPTR, HDPTR, @DOWN)
               PRMCNT = Ø
               CALL GETTKN
               IF (TOKEN .NE. @LEFTP)
                  (TOKEN .EQ. @LEFTP)
                    LOOP GIVEN LSTPTR = HDPTR
                      DO CALL FRMPRM (PRMPTR, OK)
                         IF (.NOT. OK)
                            (OK) CALL GETTKN
                              CALL JOINLL (LSTPTR, PRMPTR, @ACROSS)
                              LSTPTR = PRMPTR
```

```
                        PRMCNT = PRMCNT + 1
                    FI
                  WHILE (OK .AND. TOKEN .EQ. @COMMA)
                ENDLOOP
                IF (OK .AND. TOKEN .EQ. @RIGHTP) CALL GETTKN
                   (.NOT. OK .OR. TOKEN .NE. @RIGHTP) OK = .FALSE.
                FI
          FI
          T (1) = PRMCNT
          CALL STORLL (HDPTR, T)
          IF (.NOT. OK)
             (OK) CALL SKPEOS
          FI
       FI
       WRITE (3,*) "Leave FORMHD"
       RETURN
       END

       SUBROUTINE FRMPRM (PRMPTR,OK)
       IMPLICIT INTEGER (A-Z)
       COMMON /T/ TOKEN
       COMMON /TS/ TKNSTR (22)
       INTEGER T (2)
       LOGICAL OK
C
C A form dummy parameter is a name followed by a colon, followed
C by a parameter type.  Parameters may be either forms or integers.
C Put the name and type in the symbol table in a
C Parameter Entry.
C
       WRITE (3,*) "Enter FRMPRM"
       CALL GETTKN
       IF (TOKEN .NE. @NAME) OK = .FALSE.
          (TOKEN .EQ. @NAME)
            ADDR = NEWLOC (1)
            CALL NEWLL (PRMPTR, 2)
            CALL ADDAS (PRMNAM, TKNSTR)
            T (1) = PRMNAM
            CALL GETTKN
            IF (TOKEN .NE. @COLON) OK = .FALSE.
               (TOKEN .EQ. @COLON)
                 CALL GETTKN
                 IF (TOKEN.NE.@FORM .AND. TOKEN.NE.@INTEGER)
                       OK = .FALSE.
                    (TOKEN.EQ.@FORM .OR. TOKEN.EQ.@INTEGER)
                       T (2) = TOKEN
                       CALL STORLL (PRMPTR, T)
                 FI
            FI
```

```
      FI
      WRITE (3,*) "Leave FRMPRM"
      RETURN
      END

      SUBROUTINE STRUCT (STRPTR, OK)
      IMPLICIT INTEGER (A-Z)
      COMMON /CURPTR/ CURPTR
      COMMON /T/ TOKEN
      LOGICAL OK, EMPTY
      COMMON /DIRECT/ UP, DOWN
      LOGICAL UP, DOWN
      COMMON /S/ STRSET (10), STTSET (12), EXPSET (10)
      LOGICAL SETMEM
      WRITE (3,*) "Enter STRUCT"
C
C A structure definition has a recursive syntax.  This is the
C driver routine for the structure parsing routines.  When a
C recursive call is to be made, the calling routine pushes the
C name of the routine to be called onto the stack and returns
C to the driver, which looks at the stack and calls the
C appropriate routine.  When a return is to be made, the routine
C removes its own entry from the stack and sets the direction
C flag to UP.
C
      LOOP
        GIVEN CALL PUSH (@STRUCTURE,1); OK = .TRUE.
          DOWN = .TRUE.; UP = .FALSE.
        LOOPBY CALL LOOKAT (TOP, 1, EMPTY)
        WHILE (.NOT. EMPTY) (OK)
              (SETMEM (STRSET, TOP))
        DO
          IF
            (DOWN)
              IF
                (TOP .EQ. @STRUCTURE) CALL STRST (OK)
                (TOP .EQ. @RECORD) CALL RECST (OK)
                (TOP .EQ. @ARRAY) CALL ARRST (OK)
                (TOP .EQ. @FIELDEF) CALL FLDFST (OK)
                (TOP .EQ. @FORM) CALL FORMST (OK)
                (TOP .EQ. @PARM) CALL PARMST (OK)
              FI
            (UP)
              IF
                (TOP .EQ. @STRUCTURE) CALL STRND (OK)
                (TOP .EQ. @RECORD) CALL RECND (OK)
                (TOP .EQ. @ARRAY) CALL ARRND (OK)
                (TOP .EQ. @FIELDEF) CALL FLDFND (OK)
                (TOP .EQ. @FORM) CALL FORMND (OK)
```

```
                (TOP .EQ. @PARM) CALL PARMND (OK)
                (TOP .EQ. @EXPAND) CALL EXPAND (@FORM)
                                   CALL POP (TOP,LEN)
            FI
        FI
      ENDLOOP
      STRPTR = CURPTR
      WRITE (3,*) "Leave STRUCT"
      RETURN
      END

      SUBROUTINE STRST (OK)
      IMPLICIT INTEGER (A-Z)
      COMMON /T/ TOKEN
      COMMON /CURPTR/ CURPTR
      COMMON /CURFRM/ CURNAM, CURHDR, CURSTR, CURIMP, CUREXP
      COMMON /F/ FORSET (10)
      COMMON /DIRECT/ UP, DOWN
      COMMON /TS/ TKNSTR (22)
      COMMON /CURSIZ/ CURSIZ, CURKNO
      COMMON /SYMTAB/ FRMTAB, FRMLST, VARTAB, VARLST
      COMMON /SCAN/ FORM, DECL
      LOGICAL FORM, DECL
      INTEGER NAM (2), STR (4), S (2), T (4)
      LOGICAL OK, SETMEM, FOUND, UP, DOWN
      WRITE (3,*) "Enter STRST"
C
C A structure may be a Record, an Array, a Fortran scalar, or a
C Form.  If it is a Form it may be the Current Form, a Parameter
C to the Current Form, or a previous Form.
C
      CALL NEWLL (PTR, 4)
      CURPTR = PTR
      IF
C
C If it is a Record, make a Record Entry in the symbol table and
C make a recursive call to the record parsing routine.
C
        (TOKEN .EQ. @RECORD)
          S (1) = @RECORD; S (2) = PTR
          CALL PUSH (S, 2)
          T (1) = @RECORD; T (2) = @KNOWN; T (3) = 0; T (4) = 0
          CALL STORLL (PTR, T)
C
C If it is a Array, make a Array Entry in the symbol table and
C make a recursive call to the array parsing routine.
C
        (TOKEN .EQ. @ARRAY)
          S (1) = @ARRAY; S (2) = PTR
```

```
          CALL PUSH (S, 2)
          T (1) = @ARRAY; T (2) = @KNOWN; T (3) = Ø; T (4) = Ø
          CALL STORLL (PTR, T)
C
C If it is a Fortran scalar, make a Fortran Entry and since there
C can be no substructure, make a recursive return.
C
        (SETMEM (FORSET, TOKEN))
          T (1) = @FORTRAN; T (2) = @KNOWN; T (3) = 1; T (4) = TOKEN
          CURSIZ = 1; CURKNO = @KNOWN
          CALL STORLL (PTR, T)
          CALL GETTKN
          DOWN = .FALSE.; UP = .TRUE.
        (TOKEN .EQ. @NAME)
           CALL GETLL (CURNAM,NAM)
           RESULT = ASCMP (TKNSTR,NAM (1))
C
C If it is the current form, make a current form structure
C entry.  It must have no parameters.
C EXPAND will expand it into a type description.
C
          IF (RESULT .EQ. @EQUAL)
               T (1) = @CURRENT; T (2) = @UNKNOWN
               T (3) = Ø; T (4) = Ø
               CALL STORLL (PTR,T)
               CALL PUSH (@EXPAND,1)
               CALL EXPAND (@CURRENT)
               CALL POP (S,LEN)
               CALL GETTKN
               DOWN = .FALSE.; UP = .TRUE.
             (RESULT .NE. @EQUAL)
               CALL SRCHPM (TKNSTR,NUM,PRMPTR,FOUND)
C
C If it is a parameter to the current form, make a parameter form
C structure entry.  It must have no parameters.  Keep the
C parameter name with it for future searches.
C
               IF (FOUND)
                    T (1) = @PARM; T (2) = @UNKNOWN
                    T (3) = Ø; T (4) = NUM
                    CALL STORLL (PTR,T)
                    CALL NEWLL (PNMPTR,1)
                    CALL ADDAS (NAME,TKNSTR)
                    T (1) = NAME
                    CALL STORLL (PNMPTR,T)
                    CALL JOINLL (PTR,PNMPTR,@ACROSS)
                    CALL GETTKN
                    CURSIZ = Ø; CURKNO = @UNKNOWN
                    DOWN = .FALSE.; UP = .TRUE.
```

```
                    (.NOT. FOUND)
C
C If it is a form, it must have been previously defined.  Look
C it up and get its structure entry.  Use the size from this to
C make a Form Spec Structure entry.
                      CALL SRCHLL (@ACROSS,TKNSTR,FRMTAB,FRMPTR,FOUND)
                      IF (.NOT. FOUND) OK = .FALSE.
                         (FOUND)
                           STRPTR = FRMPTR
                           CALL NEXTLL (STRPTR,@DOWN)
                           CALL NEXTLL (STRPTR,@DOWN)
                           CALL NEXTLL (STRPTR,@ACROSS)
                           CALL GETLL (STRPTR,STR)
                           STRKNO = STR (2)
                           STRSIZ = STR (3)
                           T (1) = @FORM; T (2) = STRKNO
                           T (3) = STRSIZ; T (4) = Ø
                           CALL STORLL (PTR,T)
C
C If this structure description is part of a variable declaration
C rather than part of a Form structure, push a recursive call to
C EXPAND, to be made after all the Form parameters have been parsed.
C
                           IF (DECL) CALL PUSH (@EXPAND,1)
                              (FORM)
                           FI
                           S (1) = @FORM; S (2) = PTR
                           CALL PUSH (S,2)
                      FI
                FI
           FI
        (TOKEN .NE. @RECORD .AND. TOKEN .NE. @ARRAY .AND.
     X   TOKEN .NE. @NAME) OK = .FALSE.
      FI
      RETURN
      END

      SUBROUTINE STRND (OK)
      IMPLICIT INTEGER (A-Z)
      LOGICAL OK
      WRITE (3,*) "Enter STRND"
      CALL POP (TOP,LEN)
      RETURN
      END
```

```
      SUBROUTINE ARRST (OK)
      IMPLICIT INTEGER (A-Z)
C
C An array definition is a list of ranges, followed by a base
C structure.  The range list is not recursively defined,
C so the parsing routine for it may be called directly.
C Store the range size in the Array Entry, temporarily.
C It will be used later to calculate the size of the array.
C Make a recursive call to parse the base structure.
C
      COMMON /T/ TOKEN
      COMMON /CURPTR/ CURPTR
      INTEGER T (4)
      LOGICAL OK
      WRITE (3,*) "Enter ARRST"
      CALL RANGE (CURPTR,NUM,K,RNGSIZ,OK)
      IF (.NOT. OK)
         (OK)
           CALL GETLL (CURPTR, T)
           T (2) = K; T (3) = RNGSIZ; T (4) = NUM
           CALL STORLL (CURPTR, T)
           CALL GETTKN
           IF (TOKEN .NE. @OF) OK = .FALSE.
              (TOKEN .EQ. @OF)
                CALL PUSH (@STRUCTURE,1)
                CALL GETTKN
           FI
      FI
      RETURN
      END

      SUBROUTINE ARRND (OK)
      IMPLICIT INTEGER (A-Z)
C
C The base structure of the array has been scanned and its
C structure is pointed at by CURPTR.  Its size is in CURSIZ.
C Join the base structure to the array entry in the table.
C The range size is in the array entry.  Multiply it by the
C base size to get the total array size.  If either the range
C size or the base size is unknown at scan time, the array size
C will be unknown.
C
      COMMON /CURPTR/ CURPTR
      COMMON /CURSIZ/ CURSIZ, CURKNO
      INTEGER S (2), T (4)
      LOGICAL OK
      WRITE (3,*) "Enter ARRND"
      CALL POP (S,LEN)
      ARRPTR = S (2)
```

```
      CALL JOINLL (ARRPTR, CURPTR, @ACROSS)
      CURPTR = ARRPTR
      CALL GETLL (ARRPTR,T)
      IF (T (2) .EQ. @KNOWN .AND. CURKNO .EQ. @KNOWN)
           RANGE = T (3)
           BASE = CURSIZ
           CURSIZ = RANGE * BASE
           T (3) = BASE
      (T (2) .EQ. @UNKNOWN .OR. CURKNO .EQ. @UNKNOWN)
           IF (CURKNO .EQ. @UNKNOWN)
                T (3) = NEWLOC (1); T (2) = @UNKNOWN
              (CURKNO .EQ. @KNOWN)
                T (3) = CURSIZ; T (2) = @KNOWN
           FI
           CURSIZ = 0; CURKNO = @UNKNOWN
      FI
      CALL STORLL (ARRPTR,T)
      RETURN
      END

      SUBROUTINE RANGE (PTR, NUM, K, RNGSIZ, OK)
      IMPLICIT INTEGER (A-Z)
C
C A range list is a series of range entries separated by commas,
C enclosed in parentheses.  A range entry is a lower and upper
C limit, separated by a colon, or just an upper limit, in which
C case the lower is assumed to be 1.  Limits may be integers or
C form parameters.  If any of the limits in a range list are
C parameters, the range size cannot be calculated.
C
      COMMON /T/ TOKEN
      COMMON /TV/ TKNVAL
      COMMON /TS/ TKNSTR (22)
      COMMON /SCAN/ FORM, DECL
      LOGICAL FORM, DECL, FOUND
      INTEGER T (2)
      LOGICAL OK
      WRITE (3,*) "Enter RANGE"
      CALL GETTKN
      IF (TOKEN .NE. @LEFTP) OK = .FALSE.
         (TOKEN .EQ. @LEFTP)
           LOOP
             GIVEN
               LSTPTR = PTR; NUM = 0; RNGSIZ = 1; K = @KNOWN
             DO
               CALL GETTKN
               IF (TOKEN .EQ. @NUMBER)
                    R1 = @NUMBER; V1 = TKNVAL
                    CALL GETTKN
```

```
                (TOKEN .EQ. @NAME)
                  CALL SRCHPM (TKNSTR,PRMNUM,PRMPTR,FOUND)
                  IF (.NOT. FOUND) OK = .FALSE.
                     (FOUND)
                       R1 = @NAME; V1 = PRMNUM
                       CALL GETTKN
                  FI
                (TOKEN .NE. @NUMBER .AND. TOKEN .NE. @NAME)
                  OK = .FALSE.
              FI
              IF (.NOT. OK)
                 (OK)
                   IF (TOKEN .NE. @COLON)
                         R2 = R1; V2 = V1
                         R1 = @NUMBER; V1 = 1
                       (TOKEN .EQ. @COLON)
                         CALL GETTKN
                         IF (TOKEN .EQ. @NUMBER)
                               R2 = @NUMBER; V2 = TKNVAL
                               CALL GETTKN
                            (TOKEN .EQ. @NAME)
                              CALL SRCHPM (TKNSTR,PRMNUM,PRMPTR,FOUND)
                              IF (.NOT. FOUND) OK = .FALSE.
                                 (FOUND)
                                   R2 = @NAME; V2 = PRMNUM
                                   CALL GETTKN
                              FI
                            (TOKEN .NE. @NUMBER .AND. TOKEN .NE. @NAME)
                              OK = .FALSE.
                         FI
                    FI
                    IF (.NOT. OK)
                       (OK)
C
C If this range list is part of a Form, then each upper and lower
C bound gets stored in a Range Entry in the symbol table.
C Storage is allocated in the dope vector and the starting address
C of the range list is stored in the symbol table.
C
                              IF (FORM)
                                   ADDR = NEWLOC (1)
                                   IF (NUM .EQ. Ø)
                                        CALL NEWLL (RNGPTR,2)
                                        T (1) = @KNOWN
                                        T (2) = ADDR
                                        CALL STORLL (RNGPTR,T)
                                        CALL JOINLL (LSTPTR,RNGPTR,@DOWN)
                                        LSTPTR = RNGPTR
                                      (NUM .NE. Ø)
```

```
                                    FI
                                    CALL NEWLL (R1PTR,2)
                                    T (1) = R1; T (2) = V1
                                    CALL STORLL (R1PTR,T)
                                    CALL JOINLL (LSTPTR,R1PTR,@DOWN)
                                    CALL NEWLL (R2PTR,2)
                                    T (1) = R2; T (2) = V2
                                    ADDR = NEWLOC (1)
                                    CALL STORLL (R2PTR,T)
                                    CALL JOINLL (R1PTR,R2PTR,@DOWN)
                                    LSTPTR = R2PTR
C
C If the range list is part of a declaration, then the values are
C put in the dope vector.
C
                                (DECL)
                                  ADDR = NEWLOC (1)
                                  CALL PUTLOC (ADDR,V1)
                                  IF (NUM .EQ. Ø)
                                       CALL NEWLL (RNGPTR,2)
                                       T (1) = @KNOWN;T (2) = ADDR
                                       CALL STORLL (RNGPTR,T)
                                       CALL JOINLL (LSTPTR,RNGPTR,@DOWN)
                                     (NUM .NE. Ø)
                                  FI
                                  ADDR = NEWLOC (1)
                                  CALL PUTLOC (ADDR,V2)
                             FI
C
C If both bounds are known, check them and calculate the range size.
C
                             IF (R1 .EQ. @NUMBER .AND. R2 .EQ. @NUMBER)
                                  IF (V2 .LT. V1) OK = .FALSE.
                                     (V2 .GE. V1)
                                       IF (K .EQ. @UNKNOWN)
                                          (K .EQ. @KNOWN)
                                            RNGSIZ = RNGSIZ * (V2-V1+1)
                                       FI
                                  FI
                                (R1 .EQ. @NAME .OR. R2 .EQ. @NAME)
                                  RNGSIZ = Ø; K = @UNKNOWN
                             FI
                        FI
                   FI
                   NUM = NUM + 1
              WHILE (OK .AND. TOKEN .EQ. @COMMA)
            ENDLOOP
            IF (TOKEN .NE. @RIGHTP) OK = .FALSE.
               (TOKEN .EQ. @RIGHTP)
```

```
          FI
       FI
       RETURN
       END

       SUBROUTINE RECST (OK)
       IMPLICIT INTEGER (A-Z)
       COMMON /CURPTR/ CURPTR
       INTEGER S (5)
       LOGICAL OK
       WRITE (3,*) "Enter RECST"
C
C A record description is a list of field descriptions.
C
       S (1) = @FIELDEF; S (2) = CURPTR; S (3) = CURPTR
       S (4) = Ø; S (5) = Ø
       CALL PUSH (S,5)
       RETURN
       END

       SUBROUTINE RECND (OK)
       IMPLICIT INTEGER (A-Z)
       COMMON /T/ TOKEN
       COMMON /CURPTR/ CURPTR
       COMMON /CURSIZ/ CURSIZ, CURKNO
       INTEGER S (5), REC (4)
       LOGICAL OK
       WRITE (3,*) "Enter RECND"
C
C When a record definition is completed, put its size in CURSIZ.
C
       CALL POP (S,LEN)
       PTR = S (2)
       CALL GETLL (PTR, REC)
       IF (REC (2) .EQ. @UNKNOWN)
            CURSIZ = Ø; CURKNO = @UNKNOWN
          (REC (2) .EQ. @KNOWN)
            CURSIZ = REC (3); CURKNO = @KNOWN
       FI
       CURPTR = S (2)
       IF (TOKEN .NE. @ENDREC) OK = .FALSE.
          (TOKEN .EQ. @ENDREC) CALL GETTKN
       FI
       RETURN
       END
```

```
      SUBROUTINE FLDFST (OK)
      IMPLICIT INTEGER (A-Z)
C
C A field definition is a list of names, separated by commas,
C followed by a colon, and then a structure.
C
      COMMON /T/ TOKEN
      COMMON /CURPTR/ CURPTR
      INTEGER S (5)
      LOGICAL OK
      WRITE (3,*) "Enter FLDFST"
      CALL GETTKN
      IF (TOKEN .NE. @NAME) OK = .FALSE.
         (TOKEN .EQ. @NAME)
           LAST = CURPTR
           CALL NAMLST (FRST, LAST, NUM, OK)
           CALL POP (S, LEN)
           S (3) = LAST; S (4) = FRST; S (5) = NUM
           CALL PUSH (S, 5)
           IF (.NOT. OK)
              (OK)
                IF (TOKEN .NE. @COLON) OK = .FALSE.
                   (TOKEN .EQ. @COLON)
                     CALL GETTKN
                     CALL PUSH (@STRUCTURE,1)
                FI
           FI
      FI
      RETURN
      END

      SUBROUTINE FLDFND (OK)
      IMPLICIT INTEGER (A-Z)
C
C The structure for the current field or fields is pointed at by
C CURPTR.  Its size is in CURSIZ.  Join the structure to each
C field entry.  The offset for a field is the current record
C size.  Accumulate the record size by adding the size of each
C field to it.
C
      COMMON /T/ TOKEN
      COMMON /DIRECT/ UP, DOWN
      COMMON /CURPTR/ CURPTR
      COMMON /CURSIZ/ CURSIZ, CURKNO
      INTEGER S (5), T (4), FLD (3)
      LOGICAL UP, DOWN
      LOGICAL EOSTKN
      LOGICAL OK
      WRITE (3,*) "Enter FLDFND"
```

```
      CALL POP (S,LEN)
      RECPTR = S (2); LAST = S (3); FRST = S (4)
      NUM = S (5)
      CALL GETLL (RECPTR,T)
      NUMFLD = T (4)
      RECSIZ = T (3); RECKNO = T (2)
      STRPTR = CURPTR
      STRSIZ = CURSIZ; STRKNO = CURKNO
      LOOP GIVEN FLDPTR = FRST
        WHILE (FLDPTR .NE. Ø)
        DO
          CALL JOINLL (FLDPTR,STRPTR,@DOWN)
          CALL GETLL (FLDPTR,FLD)
          IF (RECKNO .EQ. @UNKNOWN)
               FLD (2) = @UNKNOWN
               FLD (3) = NEWLOC (1)
             (RECKNO .EQ. @KNOWN)
               FLD (2) = @KNOWN
               FLD (3) = RECSIZ
          FI
          CALL STORLL (FLDPTR,FLD)
          IF (RECKNO .EQ. @KNOWN .AND. STRKNO .EQ. @KNOWN)
               RECSIZ = RECSIZ + STRSIZ
             (RECKNO .EQ. @UNKNOWN .OR. STRKNO .EQ. @UNKNOWN)
               RECSIZ = Ø; RECKNO = @UNKNOWN
          FI
          NUMFLD = NUMFLD + 1
        LOOPBY CALL NEXTLL (FLDPTR,@ACROSS)
      ENDLOOP
      T (2) = RECKNO; T (3) = RECSIZ; T (4) = NUMFLD
      CALL STORLL (RECPTR,T)
      CURSIZ = RECSIZ; CURKNO = RECKNO
      IF (.NOT. EOSTKN (X))
         (EOSTKN (X))
           CALL SKPEOS
C
C If the next token is a name, then there is another field
C definition.  Get its name list and make a recursive call to get
C their structure.
C
           IF (TOKEN .NE. @NAME)
              (TOKEN .EQ. @NAME)
                DOWN = .TRUE.; UP = .FALSE.
                CALL NAMLST (FRST,LAST,NUM,OK)
                S (1) = @FIELDEF; S (2) = RECPTR; S (3) = LAST
                S (4) = FRST; S (5) = NUM
                CALL PUSH (S,5)
                IF (.NOT. OK)
                   (OK)
```

```
                    IF (TOKEN .NE. @COLON) OK = .FALSE.
                       (TOKEN .EQ. @COLON)
                         CALL GETTKN
                         CALL PUSH (@STRUCTURE,1)
                    FI
               FI
          FI
       FI
       RETURN
       END

       SUBROUTINE NAMLST (FRST,LAST,NUM,OK)
       IMPLICIT INTEGER (A-Z)
       COMMON /T/ TOKEN
       COMMON /TS/ TKNSTR (22)
       INTEGER T (3)
       LOGICAL OK
       WRITE (3,*) "Enter NAMLST"
C
C A name list is a list of names separated by commas.  The names
C are stored in the symbol table and joined together.
C
       LOOP
         GIVEN NUM = Ø; CALL NEWLL (PTR,3); FRST = PTR
         DO
           IF (TOKEN .NE. @NAME) OK = .FALSE.
              (TOKEN .EQ. @NAME)
                CALL ADDAS (NAME, TKNSTR)
                T (1) = NAME; T (2) = @KNOWN; T (3) = Ø
                CALL STORLL (PTR, T)
                CALL JOINLL (LAST, PTR, @ACROSS)
                LAST = PTR
                NUM = NUM + 1
                CALL GETTKN
           FI
         WHILE (OK .AND. TOKEN .EQ. @COMMA)
         LOOPBY CALL GETTKN; CALL NEWLL (PTR, 3)
       ENDLOOP
       WRITE (3,*) "Leave NAMLST"
       RETURN
       END
```

```
      SUBROUTINE FORMST (OK)
      IMPLICIT INTEGER (A-Z)
      COMMON /T/ TOKEN
           N /TS/ TKNSTR (22)
      COMMON /CURPTR/ CURPTR
      COMMON /SYMTAB/ FRMTAB, FRMLST, VARTAB, VARLST
      INTEGER T (4), S (3), FRM (3)
      LOGICAL OK, FOUND
      COMMON /DIRECT/ UP, DOWN
      LOGICAL UP, DOWN
      WRITE (3,*) "Enter FORMST"
      CALL POP (S,LEN); PTR = S (2); CALL PUSH (S,LEN)
      IF (TOKEN .NE. @NAME) OK = .FALSE.
         (TOKEN .EQ. @NAME)
C Get the number of words needed in LOCAL, from the Form Entry,
C and allocate then.  Put the form name, the LOCAL address,
C and the form pointer into the Form Spec Entry.
C If it has parameters, make a recursive call to parse them.
C
            CALL NEWLL (SPCPTR,3)
            CALL ADDAS (NAME,TKNSTR)
            CALL SRCHLL (@ACROSS,TKNSTR,FRMTAB,FRMPTR,FOUND)
            IF (.NOT. FOUND) OK = .FALSE.
               (FOUND)
                 T (1) = NAME
                 CALL GETLL (FRMPTR,FRM)
                 LOCSIZ = FRM (2)
                 ADDR = NEWLOC (LOCSIZ)
                 T (2) = ADDR; T (3) = FRMPTR
                 CALL STORLL (SPCPTR,T)
                 CALL JOINLL (PTR,SPCPTR,@DOWN)
                 CALL GETTKN
                 IF (TOKEN .EQ. @LEFTP)
                      S (1) = @PARM; S (2) = SPCPTR
                      S (3) = SPCPTR
                      CALL PUSH (S,3)
                    (TOKEN .NE. @LEFTP)
                      DOWN = .FALSE.; UP = .TRUE.
                 FI
            FI
      FI
      RETURN
      END
```

```
      SUBROUTINE FORMND (OK)
      IMPLICIT INTEGER (A-Z)
      COMMON /CURPTR/ CURPTR
      COMMON /CURSIZ/ CURSIZ, CURKNO
      INTEGER S (2), SPEC (4)
      LOGICAL OK
      WRITE (3,*) "Enter FORMND"
C
C When a Form is completely parsed, put its size in CURSIZ.
C
      CALL POP (S,LEN)
      CURPTR = S (2)
      CALL GETLL (CURPTR,SPEC)
      CURSIZ = SPEC (3); CURKNO = SPEC (2)
      RETURN
      END

      SUBROUTINE PARMST (OK)
      IMPLICIT INTEGER (A-Z)
      COMMON /T/ TOKEN
      COMMON /TV/ TKNVAL
      COMMON /CURPTR/ CURPTR
      INTEGER S (3), T (1)
      LOGICAL OK
      COMMON /DIRECT/ UP, DOWN
      LOGICAL UP, DOWN
      WRITE (3,*) "Enter PARMST"
C
C A parameter to a Form may be an integer constant or a Form
C Specification.  If it is an integer, store its value in the
C symbol table.
C
      CALL GETTKN
      CALL NEWLL (PRMPTR, 1)
      CALL POP (S, LEN)
      LSTPTR = S (3)
      CALL JOINLL (LSTPTR, PRMPTR, @ACROSS)
      S (3) = PRMPTR
      CALL PUSH (S, 3)
      IF (TOKEN .EQ. @NUMBER)
           T (1) = @INTEGER
           CALL STORLL (PRMPTR, T)
           CALL NEWLL (VALPTR, 1)
           T (1) = TKNVAL
           CALL STORLL (VALPTR, T)
           CALL JOINLL (PRMPTR, VALPTR, @DOWN)
           DOWN = .FALSE.; UP = .TRUE.
           CALL GETTKN
C
```

```
C If it is a Form Spec, make a recursive call to parse it.
C
         (TOKEN .EQ. @NAME)
           T (1) = @FORM
           CALL STORLL (PRMPTR, T)
           CURPTR = PRMPTR
           S (1) = @FORM; S (2) = PRMPTR
           CALL PUSH (S,2)
         (TOKEN .NE. @NUMBER .AND. TOKEN .NE. @NAME) OK = .FALSE.
      FI
      RETURN
      END

      SUBROUTINE PARMND (OK)
      IMPLICIT INTEGER (A-Z)
      COMMON /T/ TOKEN
      INTEGER S (3), T (2)
      LOGICAL OK
      COMMON /DIRECT/ UP, DOWN
      LOGICAL UP, DOWN
      WRITE (3,*) "Enter PARMND"
C
C If a parameter is followed by a comma, there is another
C parameter.  If it is followed by a right paren, then the
C parm list is complete.
C
      CALL POP (S,LEN)
      IF (TOKEN .EQ. @COMMA)
           DOWN = .TRUE.; UP = .FALSE.
           CALL PUSH (S, 3)
         (TOKEN .EQ. @RIGHTP)
           CALL GETTKN
         (TOKEN .NE. @COMMA .AND. TOKEN .NE. @RIGHTP) OK = .FALSE.
      FI
      RETURN
      END

      SUBROUTINE IMPEXP (IMPPTR, OK)
      IMPLICIT INTEGER (A-Z)
      LOGICAL OK
      WRITE (3,*) "Enter IMPEXP"
      CALL IMPORT (IMPPTR, OK)
      IF (.NOT. OK)
         (OK) CALL EXPORT (EXPPTR, OK)
      FI
      CALL JOINLL (IMPPTR, EXPPTR, @DOWN)
      WRITE (3,*) "Leave IMPEXP"
      RETURN
      END
```

```
      SUBROUTINE IMPORT (IMPPTR, OK)
      IMPLICIT INTEGER (A-Z)
      COMMON /T/ TOKEN
      COMMON /TS/ TKNSTR (22)
      INTEGER T (2)
      LOGICAL OK, FOUND
      WRITE (3,*) "Enter IMPORT"
C
C The Import section is a list of parameters to the current form
C each followed by a list of subroutines and functions which
C it must support.  The parameter name and number are stored in
C the symbol table and its import list is joined to it.
C
      CALL NEWLL (IMPPTR, 1); FRMCTR = Ø
      IF (TOKEN .NE. @IMPORT)
         (TOKEN .EQ. @IMPORT)
           WRITE (6,*) "In IMPORT"
           CALL GETRACE
           LOOP
             GIVEN CALL GETTKN; LAST = IMPPTR
             WHILE (TOKEN .EQ. @FROM)
                   (OK)
             DO
               CALL GETTKN
               IF (TOKEN .NE. @NAME) OK = .FALSE.
                  (TOKEN .EQ. @NAME)
                    CALL SRCHPM (TKNSTR,PRMNUM,PRMPTR,FOUND)
                    IF (.NOT. FOUND) OK = .FALSE.
                       (FOUND)
                         CALL NEWLL (FRMPTR, 3)
                         CALL JOINLL (LAST, FRMPTR, @ACROSS)
                         LAST = FRMPTR
                         CALL ADDAS (NAME, TKNSTR)
                         CALL IMPLST (FRMPTR, NUM, OK)
                         T (1) = NAME; T (2) = NUM; T (3) = PRMNUM
                         CALL STORLL (FRMPTR, T)
                         FRMCTR = FRMCTR + 1
                    FI
               FI
           ENDLOOP
      FI
      T (1) = FRMCTR
      CALL STORLL (IMPPTR, T)
      WRITE (3,*) "Leave IMPORT"
      RETURN
      END
```

```
      SUBROUTINE IMPLST (FRMPTR, NUM, OK)
      IMPLICIT INTEGER (A-Z)
      COMMON /T/ TOKEN
      COMMON /TS/ TKNSTR (22)
      COMMON /F/ FORSET (10)
      INTEGER T (2)
      LOGICAL OK
      LOGICAL EOSTKN, SETMEM
      WRITE (3,*) "Enter IMPLST"
C
C An import list is a list of subroutine and function names and
C parameter lists, and for functions, result types.  The name
C and type of each imported item is kept in the symbol table.
C In addition a separate matrix of imports and the forms that
C export them is kept for use in generating alias caller routines.
C NEWIMP and SETYPE are used to set up this matrix.
C
      LOOP
        GIVEN CALL GETTKN; LAST = FRMPTR; NUM = 0
        WHILE (TOKEN .EQ. @NAME)
              (OK)
        DO
          CALL NEWIMP (TKNSTR)
          CALL ADDAS (NAME, TKNSTR)
          CALL NEWLL (ITMPTR, 1)
          CALL JOINLL (LAST, ITMPTR, @DOWN)
          LAST = ITMPTR
          T (1) = NAME; CALL STORLL (ITMPTR, T)
          CALL NEWLL (SUBPTR, 2)
          CALL JOINLL (ITMPTR, SUBPTR, @ACROSS)
          CALL GETTKN
          IF
            (TOKEN .EQ. @LEFTP)
             CALL PRMLST (SUBPTR, NUMM, OK)
             IF (.NOT. OK)
                (OK)
                  IF (TOKEN .NE. @RIGHTP) OK = .FALSE.
                     (TOKEN .EQ. @RIGHTP)
                       CALL GETTKN
                       IF (TOKEN .NE. @COLON)
                            CALL SETYPE (@SUBROUTINE)
                            T (1) = @SUBROUTINE
                            T (2) = NUMM
                            CALL STORLL (SUBPTR, T)
                          (TOKEN .EQ. @COLON)
                            T (1) = @FUNCTION
                            T (2) = NUMM
                            CALL STORLL (SUBPTR, T)
                            CALL GETTKN
```

```
                              IF (.NOT. SETMEM (FORSET,TOKEN))
                                   OK = .FALSE.
                                 (SETMEM (FORSET,TOKEN))
                                   CALL SETYPE (TOKEN)
                                   CALL NEWLL (TYPPTR, 1)
                                   CALL JOINLL (SUBPTR, TYPPTR, @DOWN)
                                   T (1) = TOKEN
                                   CALL STORLL (TYPPTR, T)
                                   CALL GETTKN
                              FI
                         FI
                    FI
               FI
              (TOKEN .EQ. @COLON)
               T (1) = @FIELD; T (2) = Ø
               CALL STORLL (SUBPTR, T)
               CALL GETTKN
               CALL STRUCT (STRPTR, OK)
               CALL JOINLL (SUBPTR, STRPTR, @DOWN)
              (TOKEN .NE. @LEFTP .AND. TOKEN .NE. @COLON)
          FI
        LOOPBY NUM = NUM + 1
          IF (.NOT. EOSTKN (X)) OK = .FALSE.
             (EOSTKN (X)) CALL SKPEOS
          FI
      ENDLOOP
      WRITE (3,*) "Leave IMPLST"
      RETURN
      END

      SUBROUTINE PRMLST (SUBPTR, NUM, OK)
      IMPLICIT INTEGER (A-Z)
      COMMON /T/ TOKEN
      COMMON /TS/ TKNSTR (22)
      INTEGER T (1)
      LOGICAL OK
      WRITE (3,*) "Enter PRMLST"
C
C Parameter names are stored in the symbol table.  ADDPL adds the
C name to the parameter list in the Import/Export matrix.
C
      LOOP
        GIVEN CALL GETTKN; LSTPTR = SUBPTR; NUM = Ø
        DO
          IF (TOKEN .NE. @NAME) OK = .FALSE.
             (TOKEN .EQ. @NAME)
               CALL ADDPL (TKNSTR)
               CALL NEWLL (NAMPTR, 1)
               CALL ADDAS (NAME, TKNSTR)
```

```
            T (1) = NAME
            CALL STORLL (NAMPTR, T)
            CALL JOINLL (LSTPTR, NAMPTR, @ACROSS)
            LSTPTR = NAMPTR
            NUM = NUM + 1
            CALL GETTKN
        FI
      WHILE (OK) (TOKEN .EQ. @COMMA)
      LOOPBY CALL GETTKN
    ENDLOOP
    WRITE (3,*) "Leave PRMLST"
    RETURN
    END

    SUBROUTINE EXPORT (EXPPTR, OK)
    IMPLICIT INTEGER (A-Z)
    COMMON /T/ TOKEN
    COMMON /TS/ TKNSTR (22)
    COMMON /FNUM/ FNUM
    INTEGER T (2), ACTNAM (22)
    LOGICAL OK
    LOGICAL EOSTKN
    WRITE (3,*) "Enter EXPORT"
C
C The export section is a list of subroutines and functions
C supported by the form and the aliases under which they are
C being exported, for import by other forms.  Each actual/alias
C pair is stored in the symbol table and also passed to NEWEXP
C which sets up the Import/Export matrix.
C
    CALL NEWLL (EXPPTR, 1)
    LSTPTR = EXPPTR; NUM = Ø
    IF (TOKEN .NE. @EXPORT)
       (TOKEN .EQ. @EXPORT)
         LOOP
           GIVEN CALL GETTKN
           WHILE (OK) (TOKEN .EQ. @NAME)
           DO
             CALL NEWLL (ITMPTR, 2)
             CALL ADDAS (LOCNAM, TKNSTR)
             LOOP GIVEN I = 1 WHILE (I .LE. TKNSTR (1)+2)
                  DO ACTNAM (I) = TKNSTR (I) LOOPBY I = I + 1
             ENDLOOP
             CALL GETTKN
             IF (TOKEN .NE. @AS) OK = .FALSE.
                (TOKEN .EQ. @AS)
                  CALL GETTKN
                  IF (TOKEN .NE. @NAME) OK = .FALSE.
                     (TOKEN .EQ. @NAME)
```

```
                        CALL NEWEXP (FNUM,ACTNAM,TKNSTR)
                        CALL ADDAS (GENNAM, TKNSTR)
                        T (1) = GENNAM; T (2) = LOCNAM
                        CALL STORLL (ITMPTR, T)
                        CALL JOINLL (LSTPTR, ITMPTR, @ACROSS)
                        LSTPTR = ITMPTR
                        NUM = NUM + 1
                        CALL GETTKN
                        IF (.NOT. EOSTKN (X)) OK = .FALSE.
                           (EOSTKN (X)) CALL SKPEOS
                        FI
                    FI
                FI
            ENDLOOP
        FI
        WRITE (3,*) "Leave EXPORT"
        T (1) = NUM
        CALL STORLL (EXPPTR, T)
        RETURN
        END

        SUBROUTINE PROC (OK)
        IMPLICIT INTEGER (A-Z)
        COMMON /T/ TOKEN
        LOGICAL OK
        WRITE (3,*) "Enter PROC"
C
C A procedure is a heading, followed by declarations,
C followed by a body of statements.
C
        LOOP
          WHILE (TOKEN.EQ.@SUBROUTINE .OR. TOKEN.EQ.@FUNCTION .OR.
       X         TOKEN.EQ.@PROGRAM)
          DO
            CALL PROCHD (OK)
            IF (.NOT. OK)
               (OK)
                CALL DECLS (OK)
                IF (.NOT. OK)
                   (OK)
                    CALL BODY (OK)
                    IF (.NOT. OK)
                       (OK)
                        CALL SKPEOS
                    FI
                FI
            FI
        ENDLOOP
        WRITE (3,*) "Leave PROC"
```

```
RETURN
END

SUBROUTINE PROCHD (OK)
IMPLICIT INTEGER (A-Z)
COMMON /T/ TOKEN
COMMON /CURSTR/ CURSTR
COMMON /TS/ TKNSTR (22)
COMMON /NAME/ NAME (22)
COMMON /INFORM/ INFORM
LOGICAL INFORM, OK, EOSTKN
INTEGER TYPE (6), PROG (7), SUBR (1Ø), FUNC (8)
DATA TYPE /4,4,"T","Y","P","E"/
DATA SUBR /"S","U","B","R","O","U","T","I","N","E"/
DATA FUNC /"F","U","N","C","T","I","O","N"/
DATA PROG /"P","R","O","G","R","A","M"/
DATA LP,RP,COMMA /"(",")",","/
WRITE (3,*) "Enter PROCHD"
    IF (TOKEN .EQ. @SUBROUTINE) CALL NEWSTR (CURSTR,SUBR,1Ø)
       (TOKEN .EQ. @FUNCTION) CALL NEWSTR (CURSTR,FUNC,8)
       (TOKEN .EQ. @PROGRAM) CALL NEWSTR (CURSTR,PROG,7)
    FI
    CALL GETTKN
    IF (TOKEN .NE. @NAME) OK = .FALSE.
       (TOKEN .EQ. @NAME)
         CALL ADDSTR (CURSTR,TKNSTR)
         LOOP GIVEN I = 1 WHILE (I .LE. TKNSTR (1)+2)
              DO NAME (I) = TKNSTR (I) LOOPBY I = I + 1
         ENDLOOP
         CALL GETTKN
         IF (TOKEN .NE. @LEFTP)
              IF (.NOT. INFORM)
                 (INFORM)
                   CALL ADDCHR (CURSTR,LP)
                   CALL ADDSTR (CURSTR,TYPE)
                   CALL ADDCHR (CURSTR,RP)
              FI
            (TOKEN .EQ. @LEFTP)
              CALL ADDCHR (CURSTR,LP)
              IF (.NOT. INFORM)
                 (INFORM) CALL ADDSTR (CURSTR,TYPE)
              FI
              LOOP
                GIVEN CALL GETTKN
                DO IF (TOKEN .NE. @NAME) OK = .FALSE.
                      (TOKEN .EQ. @NAME)
                        CALL ADDCHR (CURSTR,COMMA)
                        CALL ADDSTR (CURSTR,TKNSTR)
                        CALL GETTKN
```

```
                    FI
                  WHILE (OK) (TOKEN .EQ. @COMMA)
                  LOOPBY CALL GETTKN
                ENDLOOP
                IF (TOKEN .NE. @RIGHTP) OK = .FALSE.
                   (TOKEN .EQ. @RIGHTP)
                     CALL ADDCHR (CURSTR,RP)
                     CALL GETTKN
                FI
            FI
            IF (.NOT. EOSTKN (X)) OK = .FALSE.
               (EOSTKN (X)) CALL SKPEOS
            FI
    FI
IF (.NOT. OK) (OK) CALL FORSTR FI
WRITE (3,*) "Leave PROCHD"
RETURN
END

SUBROUTINE DECLS (OK)
IMPLICIT INTEGER (A-Z)
COMMON /T/ TOKEN
COMMON /TS/ TKNSTR (22)
COMMON /SYMTAB/ FRMTAB, FRMLST, VARTAB, VARLST
COMMON /SCAN/ FORM, DECLL
COMMON /CURSTR/ CURSTR
COMMON /CURFRM/ FMPTR,HDPTR,STPTR,IMPTR,EXPTR
COMMON /INFORM/ INFORM
COMMON /NAME/ NAME (22)
LOGICAL FORM, DECLL, INFORM
INTEGER T (1)
LOGICAL OK
LOGICAL EOSTKN
INTEGER LP (1), RP (1), COM (1)
INTEGER INT (7), REL (4), LOG (7)
INTEGER VNAME (22), STR (5), VAR (1)
INTEGER SL (1), LOC (6), NME (5), TYPE (7), DL (11), DN (10)
INTEGER FNAME (22), ITEM (1), I2 (1), FNC (2), F2 (2), TYP (1)
LOGICAL FOUND
DATA COMMA,SLASH,SL,QUOTE /",","/","/""""/
DATA LOC /"L","O","C","A","L","("/
DATA NME /"N","A","M","E","("/
DATA TYPE /"T","Y","P","E","(","1",")"/
DATA DL /"D","A","T","A"," ","L","O","C","A","L","/"/
DATA DN /"D","A","T","A"," ","N","A","M","E","/"/
DATA LP, RP, COM /"(",")",","/
DATA INT /"I","N","T","E","G","E","R"/
DATA REL /"R","E","A","L"/
DATA LOG /"L","O","G","I","C","A","L"/
```

```
      WRITE (6,*) " In DECLS"
      CALL GETRACE
      WRITE (3,*) "Enter DECLS"
C
C Declarations are lists of variable names followed by
C a structure description.
C
      FORM = .FALSE.; DECLL = .TRUE.
      CALL SETLOC
      LOOP
        WHILE (OK) (TOKEN .EQ. @NAME)
        DO
C
C A list of names is parsed, each name being entered into
C the symbol table and joined to the last name.
C
          LOOP
            GIVEN
              CALL NEWLL (VARPTR, 1)
              FRST = VARPTR
            DO
              IF (TOKEN .NE. @NAME) OK = .FALSE.
                 (TOKEN .EQ. @NAME)
                   CALL ADDAS (VARNAM, TKNSTR)
                   T (1) = VARNAM
                   CALL STORLL (VARPTR, T)
                   CALL JOINLL (VARLST, VARPTR, @ACROSS)
                   VARLST = VARPTR
                   CALL GETTKN
              FI
            WHILE (OK) (TOKEN .EQ. @COMMA)
            LOOPBY CALL GETTKN; CALL NEWLL (VARPTR, 1)
          ENDLOOP
C
C The structure is parsed and its symbol table entry is joined
C to each variable name entry.  A Fortran declaration is
C generated for each variable.  Fortran scalars are declared
C with no change.  Any non-scalar is declared as a
C one dimensional Fortran array of Integers.
C
          IF (.NOT. OK)
             (OK)
               IF (TOKEN .NE. @COLON) OK = .FALSE.
                  (TOKEN .EQ. @COLON)
                    CALL GETTKN
                    CALL STRUCT (STRPTR, OK)
                    IF (.NOT. OK)
                       (OK)
                         LOOP GIVEN VARPTR = FRST
```

```
  DO CALL JOINLL (VARPTR, STRPTR, @DOWN)
  LOOPBY CALL NEXTLL (VARPTR, @ACROSS)
  WHILE (VARPTR .NE. Ø)
ENDLOOP
CALL GETLL (STRPTR,STR)
IF
  (STR (1) .EQ. @FORTRAN)
     DIM = Ø
     IF
       (STR (4) .EQ. @INTEGER)
         CALL NEWSTR (CURSTR,INT,7)
       (STR (4) .EQ. @REAL)
         CALL NEWSTR (CURSTR,REL,4)
       (STR (4) .EQ. @LOGICAL)
         CALL NEWSTR (CURSTR,LOG,7)
     FI
   (STR (1) .NE. @FORTRAN)
     CALL NEWSTR (CURSTR,INT,7)
     IF
       (STR (2) .EQ. @KNOWN)
         DIM = STR (3)
       (STR (2) .EQ. @UNKNOWN)
         DIM = 1
     FI
FI
LOOP
  GIVEN VARPTR = FRST
  DO
    CALL GETLL (VARPTR,VAR)
    VARNAM = VAR (1)
    CALL GETAS (VARNAM,VNAME)
    CALL EMPSTR (TEMP)
    CALL ADDSTR (TEMP,VNAME)
    IF
      (DIM .EQ. Ø)
      (DIM .NE. Ø)
        CALL NEWSTR (TEMP2,LP,1)
        D = DIM
        CALL NUMSTR (TEMP3,D)
        CALL NEWSTR (TEMP4,RP,1)
        CALL JOINST (TEMP3,TEMP4)
        CALL JOINST (TEMP2,TEMP3)
        CALL JOINST (TEMP,TEMP2)
    FI
    CALL JOINST (CURSTR,TEMP)
  LOOPBY CALL NEXTLL (VARPTR,@ACROSS)
    IF (VARPTR .EQ. Ø)
       (VARPTR .NE. Ø)
         CALL NEWSTR (TEMP,COM,1)
```

```
                                    CALL JOINST (CURSTR,TEMP)
                                FI
                              WHILE (VARPTR .NE. Ø)
                            ENDLOOP
                            CALL FORSTR
                            IF (.NOT. EOSTKN (X)) OK = .FALSE.
                               (EOSTKN (X)) CALL SKPEOS
                            FI
                      FI
                FI
          FI
      ENDLOOP
C
C Generate declarations for Imported functions
C
      LOOP
        GIVEN
          NXTFRM = IMPTR
        LOOPBY CALL NEXTLL (NXTFRM,@ACROSS)
        WHILE (NXTFRM .NE. Ø)
        DO
          LOOP
            GIVEN NXTITM = NXTFRM
            LOOPBY CALL NEXTLL (NXTITM,@DOWN)
            WHILE (NXTITM .NE. Ø)
            DO
              FNCPTR = NXTITM
              CALL NEXTLL (FNCPTR,@ACROSS)
              CALL GETLL (FNCPTR,FNC)
              IF
                (FNC (1) .NE. @FUNCTION)
                (FNC (1) .EQ. @FUNCTION)
                  CALL GETLL (NXTITM,ITEM)
                  FNCNAM = ITEM (1)
                  CALL GETAS (FNCNAM,FNAME)
                  LOOP
                    GIVEN FRM2 = IMPTR; FOUND = .FALSE.
                    LOOPBY CALL NEXTLL (FRM2,@ACROSS)
                    WHILE
                      (FRM2 .NE. NXTFRM)
                      CALL SRCHLL (@DOWN,FNAME,FRM2,ADDR,FOUND)
                      (.NOT. FOUND)
                  ENDLOOP
                  IF
                    (FOUND)
                    (.NOT. FOUND)
                      TYPTR = FNCPTR
                      CALL NEXTLL (TYPTR,@DOWN)
                      CALL GETLL (TYPTR,TYP)
```

```
                        IF
                          (TYP (1) .EQ. @INTEGER)
                            CALL NEWSTR (STR,INT,7)
                          (TYP (1) .EQ. @REAL)
                            CALL NEWSTR (STR,REL,4)
                          (TYP (1) .EQ. @LOGICAL)
                            CALL NEWSTR (STR,LOG,7)
                        FI
                        CALL ADDSTR (STR,FNAME)
                        CALL FORSTR
                    FI
                FI
            ENDLOOP
        ENDLOOP
C
C Generate declarations and data statements for dope vectors.
C
        CALL NEWSTR (CURSTR,INT,7)
        L = CURLOC (X)
        IF (L .EQ. 0)
           (L .GT. 0)
             CALL NEWSTR (TEMP,LOC,6)
             CALL JOINST (CURSTR,TEMP)
             CALL NUMSTR (TEMP,L)
             CALL ADDCHR (TEMP,RP)
             CALL ADDCHR (TEMP,COMMA)
             CALL JOINST (CURSTR,TEMP)
        FI
        CALL NEWSTR (TEMP,NME,5)
        CALL JOINST (CURSTR,TEMP)
        CALL NUMSTR (TEMP,STRLEN (NAME))
        CALL ADDCHR (TEMP,RP)
        IF (.NOT. INFORM)
           (INFORM)
             CALL ADDCHR (TEMP,COMMA)
             CALL NEWSTR (TEMP2,TYPE,7)
             CALL JOINST (TEMP,TEMP2)
        FI
        CALL JOINST (CURSTR,TEMP)
        CALL FORSTR
        L = CURLOC (X)
        IF (L .EQ. 0)
           (L .GT. 0)
             CALL NEWSTR (CURSTR,DL,11)
             CALL LOCSTR (TEMP)
             CALL JOINST (CURSTR,TEMP)
             CALL NEWSTR (TEMP,SL,1)
             CALL JOINST (CURSTR,TEMP)
             CALL FORSTR
```

```
FI
CALL NEWSTR (CURSTR,DN,10)
LOOP
  GIVEN I = 1
  DO
    CALL ADDCHR (CURSTR,QUOTE)
    CALL ADDCHR (CURSTR,NAME (I+2))
    CALL ADDCHR (CURSTR,QUOTE)
  WHILE (I .LT. NAME (1))
  LOOPBY I = I + 1
    CALL ADDCHR (CURSTR,COMMA)
ENDLOOP
CALL ADDCHR (CURSTR,SLASH)
CALL FORSTR
RETURN
END
```

```
      SUBROUTINE EXPAND (TYPE)
      IMPLICIT INTEGER (A-Z)
      COMMON /CURPTR/ CURPTR
      COMMON /CURFRM/ CURNAM, CURHDR, CURSTR, CURIMP, CUREXP
      COMMON /DIRECT/ UP, DOWN
      COMMON /EXSET/ EXSET (12)
      COMMON /CURFLG/ CURFLG
      LOGICAL CURFLG
      LOGICAL EMPTY, OK, UP, DOWN, SETMEM
      INTEGER S (4)
      WRITE (3,*) "Enter EXPAND"
C
C These routines expand a Form Specification into a type.
C This is done by tracing the Form structure and generating a
C copy of it.  Wherever a reference to a dummy parameter
C occurs in the Form, The actual parameter from the Form
C Spec is used in the copy.  Any values which were unknown
C when the Form was parsed, due to parameterisation, will
C now be known, so the dope vector for the type is generated.
C
      LOOP
        GIVEN
          IF (TYPE .EQ. @FORM)
               SPCPTR = CURPTR; CALL NEXTLL (SPCPTR,@DOWN)
               CURFLG = .FALSE.
               S (1) = @FORM; S (2) = SPCPTR; S (3) = Ø; S (4) = Ø
               CALL PUSH (S,4)
               DOWN = .TRUE.; UP = .FALSE.
             (TYPE .EQ. @CURRENT)
               CURFLG = .TRUE.
               S (1) = @CURRENT; S (2) = CURSTR; S (3) = Ø; S (4) = Ø
               CALL PUSH (S,4)
               DOWN = .TRUE.; UP = .FALSE.
          FI
        LOOPBY CALL LOOKAT (TOP,1,EMPTY)
        WHILE (.NOT. EMPTY)
              (SETMEM (EXSET,TOP))
        DO
          IF
            (DOWN)
              IF
                (TOP .EQ. @FORM) CALL XFRMST
                (TOP .EQ. @PARM) CALL XPRMST
                (TOP .EQ. @RECORD) CALL XRECST
                (TOP .EQ. @FIELD) CALL XFLDST
                (TOP .EQ. @ARRAY) CALL XARRST
                (TOP .EQ. @STRUCTURE) CALL XSTRST
                (TOP .EQ. @ACTUAL) CALL XACTST
                (TOP .EQ. @CURRENT) CALL XCURST
```

```
            FI
          (UP)
            IF
              (TOP .EQ. @FORM) CALL XFRMND
              (TOP .EQ. @PARM) CALL XPRMND
              (TOP .EQ. @RECORD) CALL XRECND
              (TOP .EQ. @FIELD) CALL XFLDND
              (TOP .EQ. @ARRAY) CALL XARRND
              (TOP .EQ. @STRUCTURE) CALL XSTRND
              (TOP .EQ. @ACTUAL) CALL XACTND
              (TOP .EQ. @CURRENT) CALL XCURND
              (TOP .EQ. @RESET) CURFLG = .TRUE.; CALL POP (TOP,LEN)
            FI
        FI
      ENDLOOP
      STRPTR = CURPTR
      RETURN
      END

      SUBROUTINE XFRMST
      IMPLICIT INTEGER (A-Z)
C
C Get the unexpanded Form Spec Entry.  This will contain
C a pointer to the form and the starting address in LOCAL.
C Get the form number from the Form Entry.  Store the starting
C address, the form pointer, and the form number in the expanded
C Form Spec Entry.  Put the form number in LOCAL.  Get the
C first unexpanded parameter pointer and make a recursive call
C to expand it.
C
      COMMON /SYMTAB/ FRMTAB, FRMLST, VARTAB, VARLST
      INTEGER XSPEC (4), S (5), SPC (3), NAM (22), FRM (3)
      LOGICAL FOUND
      WRITE (3,*) "Enter XFRMST"
      CALL POP (S,LEN)
C
C Get the unexpanded Form Spec Entry.  This will contain
C a pointer to the Form and the starting address in LOCAL.
C
      SPCPTR = S (2)
      CALL GETLL (SPCPTR,SPC)
      ADDR = SPC (2)
      FRMPTR = SPC (3)
      CALL GETLL (FRMPTR,FRM)
C
C Get the Form number from the Form Entry.  Store the Form
C number, the Form pointer and the starting address in the
C expanded Form Spec Entry.
C
```

```
      FRMNUM = FRM (3)
      XSPEC (1) = FRMPTR; XSPEC (2) = ADDR; XSPEC (3) = FRMNUM
      CALL NEWLL (XSPECP,3)
      CALL STORLL (XSPECP,XSPEC)
C
C Put the Form number in the dope vector LOCAL.
C
      CALL PUTLOC (ADDR,FRMNUM)
      S (3) = XSPECP
      CALL PUSH (S,4)
C
C Get the first unexpanded parameter pointer and make a
C recursive call to expand it.
C
      S (1) = @PARM
      PRMPTR = SPCPTR; CALL NEXTLL (PRMPTR,@ACROSS)
      S (2) = PRMPTR; S (3) = XSPECP; S (4) = XSPECP
      S (5) = 1
      CALL PUSH (S,5)
      RETURN
      END

      SUBROUTINE XFRMND
      IMPLICIT INTEGER (A-Z)
C
C There are two stages to expanding a form spec.  One is expanding
C the parameters.  The second is expanding the structure.
C
      COMMON /CURPTR/ CURPTR
      COMMON /CURSIZ/ CURSIZ, CURKNO
      COMMON /DIRECT/ UP, DOWN
      LOGICAL UP, DOWN
      INTEGER T (4), S (4), XSPEC (3)
      WRITE (3,*) "Enter XFRMND"
      CALL POP (S,LEN)
      IF (S (4) .EQ. Ø)
C
C If s (4) is Ø, then the parameters have been expanded.
C Get the expanded Form Spec Entry which contains the
C Form pointer.  Get the unexpanded structure pointer via
C the form pointer.  Make a recursive call to expand the
C structure.
C
           S (4) = 1
           CALL PUSH (S,4)
           XSPECP = S (3)
           CALL GETLL (XSPECP,XSPEC)
           FRMPTR = XSPEC (1)
           STRPTR = FRMPTR
```

```
            CALL NEXTLL (STRPTR,@DOWN)
            CALL NEXTLL (STRPTR,@DOWN)
            CALL NEXTLL (STRPTR,@ACROSS)
            S (1) = @STRUCTURE
            S (2) = STRPTR
            S (3) = XSPECP
            CALL PUSH (S,3)
            DOWN = .TRUE.; UP = .FALSE.
C
C If S (4) is 1, the the structure has been expanded and is pointed
C at by CURPTR.  S (3) has the expanded form spec pointer.  Generate
C a form structure entry.
C
         (S (4) .EQ. 1)
            T (1) = @FORM; T (2) = @KNOWN; T (3) = CURSIZ; T (4) = Ø
            CALL NEWLL (XSTRP,4)
            CALL STORLL (XSTRP,T)
            XSPECP = S (3)
            CALL JOINLL (XSTRP,XSPECP,@DOWN)
            CALL JOINLL (XSPECP,CURPTR,@DOWN)
            CURPTR = XSTRP
      FI
      RETURN
      END

      SUBROUTINE XPRMST
      IMPLICIT INTEGER (A-Z)
C
C Get the unexpanded parm pointer off the stack.  If it is Ø, the
C parm list is finished.  If it is not Ø, get the parm entry.
C This may be an Integer or a Form parameter.  If it is an
C Integer, generate an expanded parameter entry containing
C the value.  Join this to the expanded parm list.  If it is a
C Form, get its unexpanded Form Spec pointer and make a recursive
C call to expand it.
C
      COMMON /DIRECT/ UP,DOWN
      LOGICAL UP, DOWN
      INTEGER XSPEC (4), SPC (3), S (5), PRM (1), VAL (1)
      WRITE (3,*) "Enter XPRMST"
      CALL POP (S,LEN)
      PRMPTR = S (2)
      XSPECP = S (4)
      CALL GETLL (XSPECP,XSPEC)
      ADDR = XSPEC (2)
      PRMNUM = S (5)
      IF (PRMPTR .EQ. Ø)
           CALL PUSH (S,LEN)
           DOWN = .FALSE.; UP = .TRUE.
```

```
          (PRMPTR .NE. Ø)
            CALL GETLL (PRMPTR,PRM)
            IF (PRM (1) .EQ. @INTEGER)
                 PRMLST = S (3)
                 CALL NEWLL (XPRMP,1)
                 CALL STORLL (XPRMP,PRM)
                 CALL JOINLL (PRMLST,XPRMP,@ACROSS)
                 PRMLST = XPRMP
                 VALPTR = PRMPTR
                 CALL NEXTLL (VALPTR,@DOWN)
                 CALL GETLL (VALPTR,VAL)
                 CALL PUTLOC (ADDR+PRMNUM,VAL)
                 CALL NEWLL (XVALP,1)
                 CALL STORLL (XVALP,VAL)
                 CALL JOINLL (XPRMP,XVALP,@DOWN)
                 CALL NEXTLL (PRMPTR,@ACROSS)
                 S (2) = PRMPTR; S (3) = PRMLST
                 CALL PUSH (S,5)
               (PRM (1) .EQ. @FORM)
                 CALL PUSH (S,5)
                 S (1) = @FORM
                 SPCPTR = PRMPTR; CALL NEXTLL (SPCPTR,@DOWN)
                 S (2) = SPCPTR; S (3) = Ø
                 CALL PUSH (S,3)
                 CALL GETLL (SPCPTR,SPC)
                 ADDR2 = SPC (2)
                 CALL PUTLOC (ADDR+PRMNUM,ADDR2-ADDR+1)
            FI
       FI
       RETURN
       END

       SUBROUTINE XPRMND
       IMPLICIT INTEGER (A-Z)
C
C If the current parm pointer is Ø, then the parm list is finished.
C If not, then a form parm has just been expanded.  Join the
C expanded parm entry to the parm list.  Get the next unexpanded
C parm pointer and make a recursive call to expand it.
C
       COMMON /CURPTR/ CURPTR
       COMMON /DIRECT/ UP, DOWN
       LOGICAL UP, DOWN
       INTEGER S (5), XPRM (1)
       WRITE (3,*) "Enter XPRMND"
       CALL POP (S,LEN)
       PRMPTR = S (2)
       IF (PRMPTR .EQ. Ø)
          (PRMPTR .NE. Ø)
```

```
            PRMLST = S (3)
            XSTRP = CURPTR
            CALL NEWLL (XPRMP,1)
            XPRM (1) = @FORM
            CALL STORLL (XPRMP,XPRM)
            CALL JOINLL (PRMLST,XPRMP,@ACROSS)
            CALL JOINLL (XPRMP,XSTRP,@DOWN)
            PRMLST = XPRMP
            S (3) = PRMLST
            CALL NEXTLL (PRMPTR,@ACROSS)
            S (2) = PRMPTR
            PRMNUM = S (5)
            PRMNUM = PRMNUM + 1
            S (5) = PRMNUM
            CALL PUSH (S,5)
            DOWN = .TRUE.; UP = .FALSE.
      FI
      RETURN
      END

      SUBROUTINE XSTRST
      IMPLICIT INTEGER (A-Z)
C
C An unexpanded structure may be a Record, Array, Fortran scalar,
C a Form, a Form parameter, or the Current Form.
C
      COMMON /CURFRM/ CURNAM,CURHDR,CURSTR,CURIMP,CUREXP
      COMMON /CURFLG/ CURFLG
      COMMON /CURPTR/ CURPTR
      COMMON /CURSIZ/ CURSIZ, CURKNO
      COMMON /DIRECT/ UP, DOWN
      INTEGER S (4), STR (4)
      LOGICAL CURFLG, UP, DOWN
      WRITE (3,*) "Enter XSTRST"
      CALL POP (S,LEN)
      STRPTR = S (2); CALL GETLL (STRPTR,STR)
      XSPECP = S (3)
      CALL PUSH (S,LEN)
      IF
C
C If it is a Record, make a recursive call to expand it.
C
        (STR (1) .EQ. @RECORD)
          S (1) = @RECORD; S (2) = STRPTR; S (3) = Ø; S (4) = XSPECP
          CALL PUSH (S,4)
C
C If it is an Array, make a recursive call to expand it.
C
        (STR (1) .EQ. @ARRAY)
```

```
          S (1) = @ARRAY; S (2) = STRPTR; S (3) = Ø; S (4) = XSPECP
          CALL PUSH (S,4)
C
C If it is a Form, turn the current form flag off.  Get the
C unexpanded form spec entry and make a recursive call
C to expand it.
C
        (STR (1) .EQ. @FORM)
          IF (CURFLG)
               CALL PUSH (@RESET,1)
               CURFLG = .FALSE.
             (.NOT. CURFLG)
          FI
          S (1) = @FORM
          CALL NEXTLL (STRPTR,@DOWN)
          S (2) = STRPTR
          S (3) = Ø
          CALL PUSH (S,3)
C
C If it is a Fortran scalar, copy the unexpanded entry to the
C expanded entry.
C
        (STR (1) .EQ. @FORTRAN)
          CALL NEWLL (XSTRP,4)
          CALL STORLL (XSTRP,STR)
          CURPTR = XSTRP
          CURSIZ = 1
          CURKNO = @KNOWN
          DOWN = .FALSE.; UP = .TRUE.
C
C If it is a Parameter, make a recursive call to expand it.
C
        (STR (1) .EQ. @PARM)
          S (1) = @ACTPARM; S (2) = STRPTR; S (3) = Ø; S (4) = XSPECP
          CALL PUSH (S,4)
C
C If it is the current form, make a recursive call to expand it.
C
        (STR (1) .EQ. @CURRENT)
          CURFLG = .TRUE.
        S (1) = @CURRENT; S (2) = CURSTR; S (3) = Ø; S (4) = XSPECP
          CALL PUSH (S,4)
      FI
      RETURN
      END
```

```
      SUBROUTINE XSTRND
      IMPLICIT INTEGER (A-Z)
      INTEGER S (4)
      WRITE (3,*) "Enter XSTRND"
      CALL POP (S,LEN)
      RETURN
      END

      SUBROUTINE XARRST
      IMPLICIT INTEGER (A-Z)
C
C Get the pointer to the unexpanded array and generate an entry
C for the expanded array.  Expand the range entries and get the size.
C Store the size in the expanded array entry.  Get the unexpanded
C structure entry for the base type and make a recursive call
C to expand it.
C
      INTEGER S (4), T (4)
      WRITE (3,*) "Enter XARRST"
      CALL POP (S,LEN)
      STRPTR = S (2); XSPECP = S (4)
      CALL NEWLL (XSTRP,4)
      CALL XRANGE (STRPTR,XSTRP,XSPECP,RNGSIZ)
      T (1) = @ARRAY; T (2) = @KNOWN; T (3) = RNGSIZ; T (4) = Ø
      CALL STORLL (XSTRP,T)
      S (3) = XSTRP; CALL PUSH (S,4)
      S (1) = @STRUCTURE
      CALL NEXTLL (STRPTR,@ACROSS)
      S (2) = STRPTR; S (3) = XSPECP
      CALL PUSH (S,3)
      RETURN
      END

      SUBROUTINE XARRND
      IMPLICIT INTEGER (A-Z)
      COMMON /CURPTR/ CURPTR
      COMMON /CURSIZ/ CURSIZ, CURKNO
      COMMON /CURFLG/ CURFLG
      LOGICAL CURFLG
      INTEGER S (4), XSTR (4), STR (4), XSPEC (4)
      WRITE (3,*) "Enter XARRND"
      CALL POP (S,LEN)
      STRPTR = S (2); XSTRP = S (3); XSPECP = S (4)
      CALL GETLL (XSTRP,XSTR)
      CALL GETLL (STRPTR,STR)
      XSTR (4) = STR (4)
C
C If this is the current form, and the size is unknown,
C then store this fact in the expanded structure entry.
```

```
C
      IF (STR (2) .EQ. @UNKNOWN .AND. CURFLG)
           XSTR (2) = @UNKNOWN; XSTR (3) = STR (3)
           CALL STORLL (XSTRP,XSTR)
           CALL JOINLL (XSTRP,CURPTR,@ACROSS)
           CURSIZ = 0; CURKNO = @UNKNOWN; CURPTR = XSTRP
C
C If this is not the current form, or if the size is
C known, calculate the array size and store the base
C size in the expanded structure entry.
C
         (STR (2) .EQ. @KNOWN .OR. .NOT. CURFLG)
           RNGSIZ = XSTR (3)
           BASE = CURSIZ
           ARRSIZ = BASE * RNGSIZ
           XSTR (3) = BASE
           CALL STORLL (XSTRP,XSTR)
           CALL JOINLL (XSTRP,CURPTR,@ACROSS)
           CURPTR = XSTRP; CURSIZ = ARRSIZ; CURKNO = @KNOWN
C
C If the base size was previously unknown, put it in
C the dope vector.
C
           IF (STR (2) .EQ. @KNOWN)
              (STR (2) .EQ. @UNKNOWN)
                ADDR = STR (3)
                CALL GETLL (XSPECP,XSPEC)
                B = XSPEC (2)
                CALL PUTLOC (ADDR+B-1,BASE)
           FI
      FI
      RETURN
      END

      SUBROUTINE XRANGE (STRPTR,XSTRP,XSPECP,RNGSIZ)
      IMPLICIT INTEGER (A-Z)
      COMMON /CURFLG/ CURFLG
      LOGICAL CURFLG
      INTEGER RSTRT (2), XSTRT (2), XSPEC (4), RNG (2)
      WRITE (3,*) "Enter XRANGE"
C
C This routine expands the range list and puts it in the
C dope vector.  If this is the current form, then the
C starting address of the rangelist is copied from the
C unexpanded entry.  If this is not the current form then
C the starting address for the range list is determined.
C
      RNGPTR = STRPTR
      CALL NEXTLL (RNGPTR,@DOWN)
```

```
      CALL GETLL (RNGPTR,RSTRT)
      LOCADD = RSTRT (2)
      IF (CURFLG)
           ADDR = LOCADD
           XSTRT (1) = @UNKNOWN
         (.NOT. CURFLG)
           CALL GETLL (XSPECP,XSPEC)
           BASE = XSPEC (2)
           ADDR = LOCADD + BASE - 1
           XSTRT (1) = @KNOWN
      FI
      XSTRT (2) = ADDR
      CALL NEWLL (XSTRTP,2)
      CALL STORLL (XSTRTP,XSTRT)
      CALL JOINLL (XSTRP,XSTRTP,@DOWN)
C
C The routine XRNG gets the value for each bound.  These
C values are checked and the range size is accumulated, and
C they are put into the dope vector.
C
      IF (CURFLG)
         (.NOT. CURFLG)
           LOOP GIVEN RNGSIZ = 1
             LOOPBY CALL NEXTLL (RNGPTR,@DOWN)
             WHILE (RNGPTR .NE. Ø)
             DO
               CALL XRNG (RNGPTR,R1,XSPECP)
               CALL NEXTLL (RNGPTR,@DOWN)
               CALL XRNG (RNGPTR,R2,XSPECP)
               IF (R2 .LT. R1) STOP "BAD RANGE"
                       (R2 .GE. R1)
                         RNGSIZ = RNGSIZ * (R2-R1+1)
               FI
               CALL PUTLOC (ADDR,R1)
               ADDR = ADDR + 1
               CALL PUTLOC (ADDR,R2)
               ADDR = ADDR + 1
           ENDLOOP
      FI
      RETURN
      END
```

```
      SUBROUTINE XRNG (RNGPTR,RNGVAL,XSPECP)
      IMPLICIT INTEGER (A-Z)
      INTEGER RNG (2), PRM (1), VAL (1)
      WRITE (3,*) "Enter XRNG"
C
C If a bound value is a number, its value is returned.
C If it is a parameter, it is looked up in the list of
C actual parameters, which are joined to the expanded
C Form Spec Entry.  The value of the actual parameter
C is returned.
C
      CALL GETLL (RNGPTR,RNG)
      IF (RNG (1) .EQ. @NUMBER)
           RNGVAL = RNG (2)
         (RNG (1) .EQ. @NAME)
           LOOP
             GIVEN NEXTP = XSPECP; I = Ø
             WHILE (I .LT. RNG (2)) (NEXTP .NE. Ø)
             LOOPBY CALL NEXTLL (NEXTP,@ACROSS); I = I + 1
           ENDLOOP
           IF (NEXTP .EQ. Ø) STOP "NOT ENOUGH PARMS"
              (I .EQ. RNG (2))
                CALL GETLL (NEXTP,PRM)
                IF (PRM (1) .NE. @INTEGER) STOP "WRONG TYPE OF PARM"
                   (PRM (1) .EQ. @INTEGER)
                     CALL NEXTLL (NEXTP,@DOWN)
                     CALL GETLL (NEXTP,VAL)
                     RNGVAL = VAL (1)
                FI
           FI
      FI
      RETURN
      END

      SUBROUTINE XRECST
      IMPLICIT INTEGER (A-Z)
C
C Get the unexpanded record pointer.  Make an expanded Record Entry.
C Get the first unexpanded field entry and make a recursive call
C to expand it.
C
      INTEGER S (5), T (4)
      WRITE (3,*) "Enter XRECST"
      CALL POP (S,LEN)
      STRPTR = S (2); XSPECP = S (4)
      T (1) = @RECORD; T (2) = @KNOWN; T (3) = Ø; T (4) = Ø
      CALL NEWLL (XSTRP,4)
      CALL STORLL (XSTRP,T)
      S (3) = XSTRP
```

```
      CALL PUSH (S,4)
      S (1) = @FIELD
      FLDPTR = STRPTR; CALL NEXTLL (FLDPTR,@ACROSS)
      S (2) = FLDPTR; S (3) = XSTRP; S (4) = XSTRP; S (5) = XSPECP
      CALL PUSH (S,5)
      RETURN
      END

      SUBROUTINE XRECND
      IMPLICIT INTEGER (A-Z)
      COMMON /CURPTR/ CURPTR
      COMMON /CURSIZ/ CURSIZ, CURKNO
      COMMON /CURFLG/ CURFLG
      LOGICAL CURFLG
      INTEGER S (4), XSTR (4), STR (4), XSPEC (4)
      WRITE (3,*) "Enter XRECND"
C
C Get the size from the expanded structure entry.
C
      CALL POP (S,LEN)
      XSTRP = S (3); STRPTR = S (2); XSPECP = S (4)
      CALL GETLL (XSTRP,XSTR)
      CURSIZ = XSTR (3)
      CURKNO = @KNOWN
      CURPTR = XSTRP
      CALL GETLL (STRPTR,STR)
      STRKNO = STR (2)
      IF (STRKNO .EQ. @KNOWN)
         (STRKNO .EQ. @UNKNOWN)
           IF (CURFLG)
                XSTR (2) = @UNKNOWN; XSTR (3) = STR (3)
                CALL STORLL (XSTRP,XSTR)
                CURSIZ = Ø; CURKNO = @UNKNOWN; CURPTR = XSTRP
              (.NOT. CURFLG)
           FI
      FI
      RETURN
      END

      SUBROUTINE XFLDST
      IMPLICIT INTEGER (A-Z)
C
C Get the unexpanded field pointer from the stack.  Get the
C unexpanded field entry, and put the name address into the
C expanded field entry.  Store the expanded field entry.  Join
C it to the expanded field list.  Push the unexpanded structure
C pointer for the field and make a recursive call to
C expand it.
C
```

```
      INTEGER S (5), FLD (3), XFLD (3)
      WRITE (3,*) "Enter XFLDST"
      CALL POP (S,LEN)
      XSPECP = S (5); FLDPTR = S (2)
      CALL GETLL (FLDPTR,FLD)
      XFLD (1) = FLD (1); XFLD (2) = @KNOWN; XFLD (3) = Ø
      CALL NEWLL (XFLDP,3)
      CALL STORLL (XFLDP,XFLD)
      XFLSTP = S (4)
      CALL JOINLL (XFLSTP,XFLDP,@ACROSS)
      XFLSTP = XFLDP
      S (4) = XFLSTP
      CALL PUSH (S,5)
      S (1) = @STRUCTURE
      STRPTR = FLDPTR; CALL NEXTLL (STRPTR,@DOWN)
      S (2) = STRPTR; S (3) = XSPECP
      CALL PUSH (S,3)
      RETURN
      END

      SUBROUTINE XFLDND
      IMPLICIT INTEGER (A-Z)
      COMMON /CURPTR/ CURPTR
      COMMON /CURSIZ/ CURSIZ, CURKNO
      COMMON /CURFLG/ CURFLG
      COMMON /DIRECT/ UP, DOWN
      LOGICAL CURFLG, UP, DOWN
      INTEGER S (5), XREC (4), XFLD (3), FLD (3), XSPEC (4)
      WRITE (3,*) "Enter XFLDND"
      CALL POP (S,LEN)
      FLDPTR = S (2); XRECP = S (3)
      XFLSTP = S (4); XSPECP = S (5)
      CALL JOINLL (XFLSTP,CURPTR,@DOWN)
      CALL GETLL (XFLSTP,XFLD)
      CALL GETLL (FLDPTR,FLD)
C
C If this is the current form, or the size is unknown,
C store this fact in the expanded structure entry.
C
      IF (FLD (2) .EQ. @UNKNOWN .AND. CURFLG)
           XFLD (2) = @UNKNOWN; XFLD (3) = FLD (3)
           CALL STORLL (XFLSTP,XFLD)
C
C If this is not the current form, or if the size
C is known, then the offset to this field is the current
C record size.  Add the field size to the record size.
C Store the offset in the expanded Field Entry and
C store the record size in the expanded Record Entry.
C
```

```
          (FLD (2) .EQ. @KNOWN .OR. .NOT. CURFLG)
            CALL GETLL (XRECP,XREC)
            RECSIZ = XREC (3)
            OFFSET = RECSIZ
            XFLD (3) = OFFSET
            RECSIZ = RECSIZ + CURSIZ
            XREC (3) = RECSIZ
            CALL STORLL (XFLSTP,XFLD)
            CALL STORLL (XRECP,XREC)
C
C If the offset was previously unknown, put it in the
C dope vector.
C
            IF (FLD (2) .EQ. @KNOWN)
               (FLD (2) .EQ. @UNKNOWN)
                 ADDR = FLD (3)
                 CALL GETLL (XSPECP,XSPEC)
                 BASE = XSPEC (2)
                 CALL PUTLOC (ADDR+BASE-1,OFFSET)
            FI
      FI
C
C If there is another field, make a recursive call to expand it.
C
      CALL NEXTLL (FLDPTR,@ACROSS)
      IF (FLDPTR .EQ. Ø)
         (FLDPTR .NE. Ø)
           S (1) = @FIELD; S (2) = FLDPTR
           S (3) = XRECP; S (4) = XFLSTP; S (5) = XSPECP
           CALL PUSH (S,5)
           DOWN = .TRUE.; UP = .FALSE.
      FI
      RETURN
      END

      SUBROUTINE XACTST
      IMPLICIT INTEGER (A-Z)
C
C This routine expands structures which are parameters.
C The actual parameters have already been expanded so it is
C simply a matter of getting the pointer to the expanded
C structure.
C
C Get the unexpanded Structure Entry, which contains the parm number.
C Get the expanded spec entry, which points across to the
C expanded actual parameters.  Loop across until the right parm
C is reached.  Its structure has been expanded already.  Make
C a recursive return and pass it back.
C stack.
```

```
C
      COMMON /DIRECT/ UP, DOWN
      COMMON /CURFLG/ CURFLG
      LOGICAL CURFLG, UP, DOWN
      INTEGER S (4), STR (4)
      WRITE (3,*) "Enter XACTST"
      IF (CURFLG)
           DOWN = .FALSE.; UP = .TRUE.
         (.NOT. CURFLG)
           CALL POP (S,LEN)
           STRPTR = S (2); XSPECP = S (4)
           CALL GETLL (STRPTR,STR)
           PRMNUM = STR (4)
           LOOP
             GIVEN
               XPRMP = XSPECP
               I = Ø
             LOOPBY I = I + 1; CALL NEXTLL (XPRMP,@ACROSS)
             WHILE (I .LT. PRMNUM) (XPRMP .NE. Ø)
           ENDLOOP
           IF (XPRMP .EQ. Ø) STOP "NOT ENOUGH PARMS"
              (XPRMP .NE. Ø)
                XSTRP = XPRMP; CALL NEXTLL (XSTRP,@DOWN)
                S (3) = XSTRP
                CALL PUSH (S,4)
                DOWN = .FALSE.; UP = .TRUE.
           FI
      FI
      RETURN
      END

      SUBROUTINE XACTND
      IMPLICIT INTEGER (A-Z)
      COMMON /CURPTR/ CURPTR
      COMMON /CURSIZ/ CURSIZ, CURKNO
      COMMON /CURFLG/ CURFLG
      LOGICAL CURFLG
      INTEGER STR (4), XSTR (4), XSPEC (4), S (4), T (4)
      WRITE (3,*) "Enter XACTND"
C
C If this is the current form that is being expanded, it has
C no actual parameters, so generate a Parm Structure Entry
C with unknown size.
C
      CALL POP (S,LEN)
      IF (CURFLG)
           STRPTR = S (2); CALL GETLL (STRPTR, STR)
           PRMNUM = STR (4)
           CALL NEWLL (XSTRP1,4)
```

```
            T (1) = @PARM; T (2) = @UNKNOWN; T (3) = Ø; T (4) = PRMNUM
            CALL STORLL (XSTRP1,T)
            NAMPTR = STRPTR; CALL NEXTLL (NAMPTR,@ACROSS)
            CALL JOINLL (XSTRP1,NAMPTR,@ACROSS)
            CURPTR = XSTRP1; CURSIZ = Ø; CURKNO = @UNKNOWN
C
C Get the size from the expanded parameter structure.
C
         (.NOT. CURFLG)
            XSTRP = S (3); XSPECP = S (4)
            CALL GETLL (XSTRP,XSTR)
            SIZE = XSTR (3)
            CURPTR = XSTRP; CURSIZ = SIZE; CURKNO = @KNOWN
      FI
      RETURN
      END

      SUBROUTINE XCURST
      IMPLICIT INTEGER (A-Z)
      COMMON /CURFRM/ CURNAM,CURHDR,CURSTR,CURIMP,CUREXP
      INTEGER S (4), T (4)
      WRITE (3,*) "Enter XCURST"
      CALL POP (S,LEN); XSPECP = S (4)
      T (1) = @CURRENT; T (2) = @UNKNOWN; T (3) = Ø; T (4) = Ø
      CALL NEWLL (XSTRP,4)
      CALL STORLL (XSTRP,T)
      S (3) = XSTRP
      CALL PUSH (S,4)
      S (1) = @STRUCTURE; S (2) = CURSTR; S (3) = XSPECP
      CALL PUSH (S,3)
      RETURN
      END

      SUBROUTINE XCURND
      IMPLICIT INTEGER (A-Z)
      COMMON /CURPTR/ CURPTR
      INTEGER S (4)
      WRITE (3,*) "Enter XCURND"
      CALL POP (S,LEN)
      XSTRP = S (3)
      CALL JOINLL (XSTRP,CURPTR,@DOWN)
      CURPTR = XSTRP
      RETURN
      END
```

```
      SUBROUTINE REFST (OK)
      IMPLICIT INTEGER (A-Z)
      COMMON /CURSTR/ CURSTR
      COMMON /T/ TOKEN
      COMMON /TS/ TKNSTR (22)
      COMMON /DIRECT/ UP, DOWN
      COMMON /SYMTAB/ FRMTAB, FRMLST, VARTAB, VARLST
      COMMON /CURFRM/ CURNAM, CURHDR, CURST, CURIMP, CUREXP
      COMMON /CALL/ CALL
      COMMON /REFLOC/ REFLOC
      COMMON /REFTYP/ REFTYP, TYPSET
      LOGICAL CALL, TYPSET
      INTEGER S (6), STR (4)
      LOGICAL FOUND, OK, UP, DOWN
C
C TKNSTR contains the current variable name, look it up.
C If it is found, it has a Var Name Entry in the symbol table.
C The Var Name Entry points down to a Structure Entry.
C Get this and put the pointers to the Var Name Entry and
C its Structure Entry onto the stack.  A ref may be followed
C by a subscript list, a parameter list, or a field name.
C This may be determined by the next token.  REFGN1 generates
C the necessary stack entries if it is a left paren, and REFGN2
C does the same for a period.  If the reference is not followed
C by either a paren or a period, then the reference is complete.
C
      WRITE (3,*) "Enter REFST"
      CALL = .FALSE.
      TYPSET = .FALSE.
      CALL POP (S,LEN)
      S (6) = REFLOC
      CALL SRCHLL (@ACROSS, TKNSTR, VARTAB, VARPTR, FOUND)
      IF (FOUND)
           STRPTR = VARPTR
           CALL NEXTLL (STRPTR,@DOWN)
           CALL GETLL (STRPTR, STR)
           S (3) = VARPTR; S (4) = STRPTR
           CALL GETTKN
           IF (TOKEN .EQ. @LEFTP) CALL REFGN1 (S, STRPTR, STR, OK)
              (TOKEN .EQ. @PERIOD) CALL REFGN2 (S, STRPTR, STR, OK)
                IF (.NOT. OK) (OK) CALL GETTKN FI
              (TOKEN .NE. @LEFTP .AND. TOKEN .NE. @PERIOD)
                S (2) = @REFERENCE; CALL EMPSTR (S(5))
                CALL PUSH (S,6)
                CALL EMPSTR (CURSTR)
                DOWN = .FALSE.; UP = .TRUE.
           FI
C
C If the name was not found, it may be an integer parameter
```

```
C to the current form.  If it is found in the parameter list,
C put its parm number on the stack.
C
         (.NOT. FOUND)
            CALL SRCHPM (TKNSTR,NUM,PRMPTR,FOUND)
            IF (FOUND)
                 S (2) = @PARM
                 S (3) = NUM
                 CALL PUSH (S,5)
                 CALL GETTKN
                 DOWN = .FALSE.; UP = .TRUE.
C
C If the name was not found, assume that it is an undeclared
C Fortran variable, function or subroutine.  It may still be
C followed by a paren, and if it is, this must contain a
C parameter list.
C
              (.NOT. FOUND)
                 CALL ADDAS (NAME, TKNSTR)
                 S (3) = @FORTRAN; S (4) = NAME
                 CALL GETTKN
                 IF (TOKEN .EQ. @LEFTP)
                           S (2) = @PARMLIST; CALL PUSH (S,6)
                           S (1) = @PARMLIST; CALL EMPSTR (S (2))
                           CALL PUSH (S,2)
                    (TOKEN .NE. @LEFTP)
                           S (2) = Ø; CALL PUSH (S,6)
                           DOWN = .FALSE.; UP = .TRUE.
                 FI
            FI
      FI
      RETURN
      END

      SUBROUTINE REFND (OK)
      IMPLICIT INTEGER (A-Z)
      COMMON /CURSTR/ CURSTR
      COMMON /T/ TOKEN
      COMMON /DIRECT/ UP, DOWN
      COMMON /CALL/ CALL
      COMMON /REFLOC/ REFLOC
      INTEGER TAIL (2), S (6), STR (4), VAR (2), VARNAM (22)
      LOGICAL CALL, OK, UP, DOWN
      INTEGER TYPE (5)
      DATA TYPE /"T","Y","P","E","("/
      DATA TAIL /"1",")"/, LB /"("/, RB /")"/
C
C The Reference Entry on the stack contains an indication of
C which structure has just been parsed.  If it was a subscript
```

```
C list, it may be followed by a further subscript, or a field.
C REFGN1 and REFGN2 will generate the necessary stack entries.
C If it is not followed by a subscript or field, generate
C an access expression
C          varname (access+1)
C The routine WRAP determines whether a type changing function
C must be wrapped around the reference.
C
C
      WRITE (3,*) "Enter REFND"
      CALL POP (S,LEN); LAST =  S (2)
      CALL JOINST (S (5),CURSTR)
      IF (LAST .EQ. @SUBSCRIPT)
           IF (TOKEN .NE. @RIGHTP) OK = .FALSE.
              (TOKEN .EQ. @RIGHTP)
                STRPTR = S (4)
                CALL NEXTLL (STRPTR,@ACROSS)
                S (4) = STRPTR
                CALL GETLL (STRPTR, STR)
                CALL GETTKN
                IF (TOKEN .EQ. @LEFTP)
                     CALL REFGN1 (S, STRPTR, STR, OK)
                     IF (.NOT. OK)
                        (OK) DOWN = .TRUE.; UP = .FALSE.
                     FI
                   (TOKEN .EQ. @PERIOD)
                     CALL REFGN2 (S, STRPTR, STR, OK)
                     IF (.NOT. OK)
                        (OK) DOWN = .TRUE.; UP = .FALSE.;
                             CALL GETTKN
                     FI
                   (TOKEN .NE. @LEFTP .AND. TOKEN .NE. @PERIOD)
                     VARPTR = S (3)
                     CALL GETLL (VARPTR, VAR)
                     CALL GETAS (VAR (1), VARNAM)
                     CALL EMPSTR (CURSTR)
                     CALL ADDSTR (CURSTR,VARNAM)
                     CALL ADDCHR (CURSTR,LB)
                     CALL JOINST (CURSTR,S (5))
                     CALL NEWSTR (TEMP, TAIL, 2)
                     CALL JOINST (CURSTR,TEMP)
                     REFLOC = S (6)
                     CALL WRAP
                FI
           FI
C
C If the last structure was a field, then the reference is
C complete.  Generate an access expression as above.
C
```

```
          (LAST .EQ. @FIELD)
            IF (CALL) CALL = .FALSE.
               (.NOT. CALL)
            VARPTR = S (3)
            CALL GETLL (VARPTR, VAR)
            CALL GETAS (VAR (1), VARNAM)
                 CALL EMPSTR (CURSTR)
                 CALL ADDSTR (CURSTR,VARNAM)
                 CALL ADDCHR (CURSTR,LB)
                 CALL JOINST (CURSTR,S (5))
                 CALL NEWSTR (TEMP, TAIL, 2)
                 CALL JOINST (CURSTR,TEMP)
                 REFLOC = S (6)
                 CALL WRAP
            FI
C
C If the last structure was a reference, then there were
C no subscripts or fields.  The routine TYPREF looks at the
C structure and if it is a Fortran scalar, records it.
C If this is a Fortran scalar reference, then the access
C expression is varname, if not it is varname (1).
C
          (LAST .EQ. @REFERENCE)
                 VARPTR = S (3)
                 STRPTR = VARPTR; CALL NEXTLL (STRPTR,@DOWN)
                 CALL TYPREF (STRPTR)
                 CALL GETLL (VARPTR, VAR)
                 CALL GETAS (VAR (1), VARNAM)
                 CALL EMPSTR (CURSTR)
                 CALL ADDSTR (CURSTR,VARNAM)
                 CALL GETLL (STRPTR,STR)
                 REFLOC = S (6)
                 IF (STR (1) .EQ. @FORTRAN)
                    (STR (1) .NE. @FORTRAN)
                      CALL ADDCHR (CURSTR,LB)
                      CALL JOINST (CURSTR,S (5))
                      CALL NEWSTR (TEMP, TAIL, 2)
                      CALL JOINST (CURSTR,TEMP)
                     CALL WRAP
                 FI
C
C If the last structure was a parmlist, then the
C access expression has already been generated.
C
          (LAST .EQ. @PARMLIST)
            IF (TOKEN .NE. @RIGHTP) OK = .FALSE.
               (TOKEN .EQ. @RIGHTP) CALL GETTKN
            FI
          (LAST .EQ. Ø)
```

```
C
C If the last structure was a Ø, then this is an
C undeclared Fortran scalar, so just generate its name.
C
           NAME = S (4)
           CALL GETAS (NAME, VARNAM)
           CALL EMPSTR (CURSTR)
           CALL ADDSTR (CURSTR,VARNAM)
C
C If the last structure was a parameter then this is an
C integer parameter to the current form.  Its value will
C be in the passed dope vector TYPE so generate
C          TYPE (parm#+1)
C
         (LAST .EQ. @PARM)
           CALL NEWSTR (CURSTR,TYPE,5)
           PRMNUM = S (3)
           P = PRMNUM + 1
           CALL NUMSTR (TEMP,P)
           CALL ADDCHR (TEMP,RB)
           CALL JOINST (CURSTR,TEMP)
      FI
      RETURN
      END

      SUBROUTINE REFGN1 (S, STRPTR, STR, OK)
      IMPLICIT INTEGER (A-Z)
      COMMON /TS/ TKNSTR (22)
      COMMON /CURFRM/ CURNAM, CURHDR, CURSTR, CURIMP, CUREXP
      INTEGER STR (4), FRM (3), S (6)
      LOGICAL OK
C
C REFGN1 generates stack entries for the situation where a
C reference or a field is followed by a left parenthesis.
C This may indicate a subscript list, or a parameter list.
C This cannot be determined syntactically, but may be
C determined semantically, based on the structure of the
C preceding reference or field.
C
C If the structure is an array, this must be a subscript list.
C
      WRITE (3,*) "Enter REFGN1"
      IF (STR (1) .EQ. @ARRAY)
           S (2) = @SUBSCRIPT; CALL PUSH (S,6)
           S (1) = @SUBSCRIPT; S (2) = @ARRAY; S (3) = STRPTR
           CALL EMPSTR (S (4))
           CALL PUSH (S,4)
C
C If the structure is a form, this must be a subscript list.
```

```
C Form structure has been expanded. Get form pointer from spec
C entry and structure pointer is below that.
C
         (STR (1) .EQ. @FORM)
           PTR = STRPTR
           CALL NEXTLL (PTR,@DOWN)
           CALL GETLL (PTR, FRM)
           FRMPTR = FRM (1)
           CALL NEXTLL (PTR,@DOWN)
           STRPTR = PTR
           S (4) = STRPTR
           S (2) = @SUBSCRIPT
           CALL PUSH (S,6)
           S (1) = @SUBSCRIPT; S (2) = @FORM; S (3) = FRMPTR
           CALL EMPSTR (S (4))
           CALL PUSH (S,4)
C
C If the structure is the current form, this must be a subscript list.
C Current structure is expanded.  Use current form pointer and
C get structure pointer below current structure pointer.
C
         (STR (1) .EQ. @CURRENT)
           FRMPTR = CURNAM
           CALL NEXTLL (STRPTR,@DOWN)
           S (4) = STRPTR
           S (2) = @SUBSCRIPT
           CALL PUSH (S,6)
           S (1) = @SUBSCRIPT; S (2) = @CURRENT; S (3) = FRMPTR
           CALL EMPSTR (S (4))
           CALL PUSH (S,4)
C
C If the structure is a Fortran scalar, this must be a parameter list.
C
         (STR (1) .EQ. @FORTRAN)
           CALL ADDAS (NAME, TKNSTR)
           S (2) = @PARMLIST; S (3) = @FORTRAN; S (4) = NAME
           CALL PUSH (S,6)
           S (1) = @PARMLIST; CALL EMPSTR (S (2))
           CALL PUSH (S,2)
         (STR (1) .EQ. @RECORD .OR. STR (1) .EQ. @PARM) OK = .FALSE.
      FI
      RETURN
      END
```

```
      SUBROUTINE REFGN2 (S, STRPTR, STR, OK)
      IMPLICIT INTEGER (A-Z)
      COMMON /CURFRM/ CURNAM, CURHDR, CURSTR, CURIMP, CUREXP
      INTEGER PRM (4), NAMSTR (22), S (6), STR (4), FRM (3), T (1)
      LOGICAL OK, FOUND
C
C REFGN2 generates stack entries for the situation where a
C reference or a field is followed by a period.  This may indicate
C a field name or a subroutine or function name.  This can be
C determined semantically from the structure type of the
C preceding reference or field.
C
C If the structure is a record, this must be a field.
C
      WRITE (3,*) "Enter REFGN2"
      IF (STR (1) .EQ. @RECORD)
           S (2) = @FIELD; CALL PUSH (S,6)
           S (1) = @FIELD; S (2) = @RECORD; S (3) = Ø; S (4) = STRPTR
           CALL EMPSTR (S (5))
           CALL PUSH (S,6)
C
C If the structure is a previously defined form, or the current
C form, this may be a field, or a subroutine or function.  Put a
C pointer to the form on the stack, so that the field parsing
C routine can determine which.
C
         (STR (1) .EQ. @FORM)
           PTR = STRPTR; CALL NEXTLL (PTR,@DOWN)
           SPCPTR = PTR
           CALL NEXTLL (PTR,@DOWN)
           STRPTR = PTR
           S (4) = STRPTR
           S (2) = @FIELD
           CALL PUSH (S,6)
           S (1) = @FIELD; S (2) = @FORM; S (3) = STRPTR; S (4) = SPCPTR
           CALL EMPSTR (S (5))
           CALL PUSH (S,6)
         (STR (1) .EQ. @CURRENT)
           FRMPTR = CURNAM
           CALL NEXTLL (STRPTR,@DOWN)
           S (4) = STRPTR
           S (2) = @FIELD
           CALL PUSH (S,6)
           S(1) = @FIELD; S(2) = @CURRENT; S(3) = STRPTR; S(4) = FRMPTR
           CALL EMPSTR (S (5))
           CALL PUSH (S,6)
C
C If the structure is a parameter form, then this must be an
C imported subroutine or function.  Find the form name in the
```

```
C import list, so that the field parser can look up the
C routine name.
C
         (STR (1) .EQ. @PARM)
           S (2) = @FIELD; CALL PUSH (S,6)
           CALL GETLL (STRPTR,PRM)
           PRMNUM = PRM (4)
           PRMPTR = STRPTR; CALL NEXTLL (PRMPTR,@ACROSS)
           CALL GETLL (PRMPTR,T)
           NAMADD = T (1)
           CALL GETAS (NAMADD, NAMSTR)
           CALL SRCHLL (@ACROSS, NAMSTR, CURIMP, LSTPTR, FOUND)
           IF (.NOT. FOUND) OK = .FALSE.
              (FOUND) S (1) = @FIELD; S (2) = @PARM
                      S (3) = PRMNUM; S (4) = LSTPTR
                      CALL EMPSTR (S (5))
                      CALL PUSH (S,6)
           FI
         (STR (1) .EQ. @FORTRAN .OR. STR (1) .EQ. @ARRAY) OK = .FALSE.
       FI
       RETURN
       END

       SUBROUTINE FLDST (OK)
       IMPLICIT INTEGER (A-Z)
       COMMON /T/ TOKEN
       COMMON /TS/ TKNSTR (22)
       COMMON /DIRECT/ UP, DOWN
       INTEGER FLDNAM (22), S (6), STR (4), EXP (2), IMPFRM (3), FLD (3)
       INTEGER IMP (2)
       LOGICAL OK, FOUND, UP, DOWN
       INTEGER SPC (3), TYPE (4), LOCAL (6), TYPE2 (1Ø)
       DATA TYPE /"T","Y","P","E"/
       DATA LOCAL /"L","O","C","A","L","("/
       DATA RIGHT /")"/
       DATA TYPE2 /"T","Y","P","E","(","T","Y","P","E","("/
C
C TKNSTR holds the current field name.  The stack entry holds
C the type of the last structure parsed, and an appropriate
C pointer.  This 'field' may be a field or a subroutine or
C function reference.
C
       WRITE (3,*) "Enter FLDST"
       CALL POP (S,LEN)
       LAST = S (2)
       PTR = S (4)
C
C If the last structure was a record, then the pointer is to
C the head of the field list.  Search for the field name and
```

```
C get its structure.  Put this info back on the stack.  This
C field may be followed by another field or a subscript.  This
C can be determined by looking at the next token.  REFGN1 and
C REFGN2 will set up the necessary stack entries.
C
      IF (LAST .EQ. @RECORD)
           CALL SRCHLL (@ACROSS, TKNSTR, PTR, FLDPTR, FOUND)
           IF (.NOT. FOUND) OK = .FALSE.
              (FOUND)
                S (3) = FLDPTR
                STRPTR = FLDPTR; CALL NEXTLL (STRPTR,@DOWN)
                CALL GETLL (STRPTR, STR)
                S (4) = STRPTR
                CALL GETTKN
                IF (TOKEN .EQ. @LEFTP) CALL REFGN1 (S, STRPTR, STR, OK)
                   (TOKEN .EQ. @PERIOD) CALL REFGN2 (S, STRPTR, STR, OK)
                     IF (.NOT. OK) (OK) CALL GETTKN FI
                   (TOKEN .NE. @LEFTP .AND. TOKEN .NE. @PERIOD)
                     S (2) = Ø; CALL EMPSTR (S(5))
                     CALL PUSH (S,6)
                     CALL EMPSTR (CURSTR)
                     DOWN = .FALSE.; UP = .TRUE.
                FI
           FI
C
C If the last structure was a parameter form, then the pointer
C is to the head of the import list.  Look up the routine name,
C and put a pointer to it on the stack.  It will be followed by
C a parameter list.
C A call to an imported subroutine or function must pass
C the dope vector of the variable through which it is
C called.  This is contained in the dope vector TYPE but
C must be referenced indirectly through a pointer which
C is also in TYPE.  This is because different instances
C of a form may have different length dope vectors.
C So generate TYPE(TYPE(parm#+1))
C
          (LAST .EQ. @PARM)
            CALL SRCHLL (@DOWN, TKNSTR, PTR, ENTPTR, FOUND)
            IF (.NOT. FOUND) OK = .FALSE.
               (FOUND)
                 PRMNUM = S (3)
                 S (3) = @PARM
                 S (2) = @PARMLIST
                 CALL GETLL (ENTPTR,IMP)
                 NAME = IMP (1); S (4) = NAME
                 CALL EMPSTR (S (5))
                 CALL PUSH (S,6)
                 S (1) = @PARMLIST
```

```
                CALL GETTKN
                CALL NEWSTR (S (2), TYPE2, 1Ø)
                P = PRMNUM + 1
                CALL NUMSTR (TEMP,P)
                CALL ADDCHR (TEMP,RIGHT)
                CALL ADDCHR (TEMP,RIGHT)
                CALL JOINST (S (2), TEMP)
                CALL PUSH (S,2)
                WRITE (3,1Ø01) (TKNSTR(I+2),I=1,TKNSTR(1))
 1Ø01           FORMAT (45X,"Generate call to ",20A1)
           FI
C
C If the last structure was a previously defined form, or the
C current form, then this may be a field name or a subroutine
C or function name.  Get the structure of the form and if it is
C a record, look for the field name.  If it is there, get its
C structure and put this info on the stack.  It may be followed
C by a further field or subscript list.  This may be determined
C by looking at the next token.  REFGN1 and REFGN2 will set up
C the necessary stack entries.
C
          (LAST .EQ. @FORM .OR. LAST .EQ. @CURRENT)
            STRPTR = S (3)
            CALL GETLL (STRPTR, STR)
            IF (STR (1) .EQ. @RECORD)
                 CALL SRCHLL (@ACROSS, TKNSTR, STRPTR, FLDPTR, FOUND)
               (STR (1) .NE. @RECORD) FOUND = .FALSE.
            FI
            IF (FOUND)
                 CALL GETLL (FLDPTR, FLD)
                 CALL GETAS (FLD (1), FLDNAM)
                 S (3) = FLDPTR
                 STRPTR = FLDPTR; CALL NEXTLL (STRPTR,@DOWN)
                 CALL GETLL (STRPTR, STR)
                 S (4) = STRPTR
                 CALL GETTKN
                 IF (TOKEN .EQ. @LEFTP) CALL REFGN1 (S, STRPTR, STR, OK)
                    (TOKEN .EQ. @PERIOD) CALL REFGN2 (S, STRPTR, STR, OK)
                      IF (.NOT. OK) (OK) CALL GETTKN FI
                    (TOKEN .NE. @LEFTP .AND. TOKEN .NE. @PERIOD)
                      CALL PUSH (S,6)
                      DOWN = .FALSE.; UP = .TRUE.
                 FI
C
C If the form structure is not a record, or the name is not in
C the field list, then this must be a subroutine or function
C reference.  It may be a reference to an exported or a local name.
C Look for it in the export list.  If it is there, get the local
C name, else assume that it is a local name already.
```

```
C The dope vector of the variable through which it is being
C called must be passed.  If its type is the current form
C then pass the dope vector TYPE, else pass the appropriate
C part of the dope vector LOCAL.
C
              (.NOT. FOUND)
                WRITE (3,1001) (TKNSTR(I+2),I=1,TKNSTR(1))
                IF (LAST .EQ. @FORM)
                     SPCPTR = PTR
                     CALL GETLL (SPCPTR,SPC)
                     ADDR = SPC (2)
                     FRMPTR = SPC (1)
                     CALL NEWSTR (TEMP,LOCAL,6)
                     CALL NUMSTR (TEMP2,ADDR)
                     CALL ADDCHR (TEMP2,RIGHT)
                     CALL JOINST (TEMP,TEMP2)
                   (LAST .EQ. @CURRENT)
                     FRMPTR = PTR
                     CALL NEWSTR (TEMP,TYPE,4)
                FI
                NEXT = FRMPTR
                CALL NEXTLL (NEXT,@DOWN)
                CALL NEXTLL (NEXT,@DOWN)
                CALL NEXTLL (NEXT,@DOWN)
                CALL NEXTLL (NEXT,@DOWN); EXPPTR = NEXT
                CALL SRCHLL (@ACROSS, TKNSTR, EXPPTR, ENTPTR, FOUND)
                IF (FOUND) CALL GETLL (ENTPTR, EXP); NAME = EXP (2)
                   (.NOT. FOUND) CALL ADDAS (NAME, TKNSTR)
                FI
                S (2) = @PARMLIST; S (3) = @FORM; S (4) = NAME
                CALL EMPSTR (S (5))
                CALL PUSH (S,6)
                S (1) = @PARMLIST; S (2) = TEMP
                CALL PUSH (S,2)
                CALL GETTKN
          FI
     FI
     RETURN
     END
```

```
      SUBROUTINE FLDND (OK)
      IMPLICIT INTEGER (A-Z)
      COMMON /T/ TOKEN
      COMMON /DIRECT/ UP, DOWN
      COMMON /CURSTR/ CURSTR
      INTEGER S (6), STR (4), FLD (3), FLDNAM (22)
      INTEGER TYPEOF (5), TAIL (2), RIGHT (1)
      LOGICAL OK, UP, DOWN
      DATA TYPEOF /"T","Y","P","E","("/
      DATA TAIL /")","+"/, PLUS /"+"/
      DATA LEFT /"("/, RIGHT /")"/
      WRITE (3,*) "Enter FLDND"
C
C The Field Entry on the stack contains an indication of
C structure has just been passed.  If it was a subscript
C list, it may be followed by another subscript list or
C a field.  REFGN1 and REFGN2 set up the necessary stack
C entries for these cases.  If it is not followed by a
C subscript or a field, then generate an access expression.
C
      CALL POP (S,LEN)
      CALL JOINST (S (5),CURSTR)
      IF (S (2) .EQ. @SUBSCRIPT)
           IF (TOKEN .NE. @RIGHTP) OK = .FALSE.
              (TOKEN .EQ. @RIGHTP)
                STRPTR = S (4)
                CALL NEXTLL (STRPTR,@ACROSS)
                S (4) = STRPTR
                CALL GETLL (STRPTR, STR)
                CALL GETTKN
                IF (TOKEN .EQ. @LEFTP)
                     CALL REFGN1 (S, STRPTR, STR, OK)
                     IF (.NOT. OK)
                        (OK) DOWN = .TRUE.; UP = .FALSE.
                     FI
                   (TOKEN .EQ. @PERIOD)
                     CALL REFGN2 (S, STRPTR, STR, OK)
                     IF (.NOT. OK)
                        (OK) DOWN = .TRUE.; UP = .FALSE.;
                             CALL GETTKN
                     FI
C
C  The offset to the field is in the symbol table.  If it
C is known then generate the value and join it to the rest
C of the current access expression.  If it is unknown then
C it will be in the dope vector TYPE, so generate
C          TYPE(offsetaddress)+
C
                    (TOKEN .NE. @LEFTP .AND. TOKEN .NE. @PERIOD)
```

```
                    FLDPTR = S (3); CALL GETLL (FLDPTR,FLD)
                    K = FLD (2)
                    IF (K .EQ. @KNOWN)
                         OFFSET = FLD (3)
                         CALL NUMSTR (CURSTR,OFFSET)
                         CALL ADDCHR (CURSTR,PLUS)
                       (K .EQ. @UNKNOWN)
                         ADDR = FLD (3)
                         CALL NEWSTR (CURSTR,TYPEOF,5)
                         CALL NUMSTR (TEMP,ADDR)
                         CALL JOINST (CURSTR,TEMP)
                         CALL NEWSTR (TEMP,TAIL,2)
                         CALL JOINST (CURSTR,TEMP)
                    FI
                    CALL JOINST (CURSTR,S (5))
               FI
          FI
C
C If the last structure was a field then generate an access
C expression for it as above.  The routine TYPREF looks at
C the type of the current field and records it if it is
C a Fortran scalar.
C
        (S (2) .EQ. @FIELD)
          FLDPTR = S (3); CALL GETLL (FLDPTR,FLD)
          STRPTR = FLDPTR; CALL NEXTLL (STRPTR,@DOWN)
          CALL TYPREF (STRPTR)
          K = FLD (2)
          IF (K .EQ. @KNOWN)
               OFFSET = FLD (3)
               CALL NUMSTR (CURSTR,OFFSET)
               CALL ADDCHR (CURSTR,PLUS)
             (K .EQ. @UNKNOWN)
               ADDR = FLD (3)
               CALL NEWSTR (CURSTR,TYPEOF,5)
               CALL NUMSTR (TEMP,ADDR)
               CALL JOINST (CURSTR,TEMP)
               CALL NEWSTR (TEMP,TAIL,2)
               CALL JOINST (CURSTR,TEMP)
          FI
          CALL JOINST (CURSTR,S (5))
C
C If the last structure was a parm list then generate
C      name(parmlist)
C
        (S (2) .EQ. @PARMLIST)
          IF (TOKEN .NE. @RIGHTP) OK = .FALSE.
             (TOKEN .EQ. @RIGHTP)
               CALL GETTKN
```

```
                CALL GETAS (S (4), FLDNAM)
                CALL EMPSTR (TEMP)
                CALL ADDSTR (TEMP,FLDNAM)
                CALL ADDCHR (TEMP,LEFT)
                CALL JOINST (TEMP,CURSTR)
                CURSTR = TEMP
                CALL NEWSTR (TEMP,RIGHT,1)
                CALL JOINST (CURSTR,TEMP)
            FI
C
C If the last structure was none of the above then generate
C an access expression as above but there will be no
C accumulated access expression to join it to.
C
          (S (2) .NE. @SUBSCRIPT .AND. S (2) .NE. @FIELD .AND.
     X     S (2) .NE. @PARMLIST)
            FLDPTR = S (3); CALL GETLL (FLDPTR,FLD)
            STRPTR = FLDPTR; CALL NEXTLL (STRPTR,@DOWN)
            CALL TYPREF (STRPTR)
            K = FLD (2)
            IF (K .EQ. @KNOWN)
                OFFSET = FLD (3)
                CALL NUMSTR (CURSTR,OFFSET)
                CALL ADDCHR (CURSTR,PLUS)
              (K .EQ. @UNKNOWN)
                ADDR = FLD (3)
                CALL NEWSTR (CURSTR,TYPEOF,5)
                CALL NUMSTR (TEMP,ADDR)
                CALL JOINST (CURSTR,TEMP)
                CALL NEWSTR (TEMP,TAIL,2)
                CALL JOINST (CURSTR,TEMP)
            FI
        FI
        RETURN
        END

        SUBROUTINE PLSTST (OK)
        IMPLICIT INTEGER (A-Z)
        COMMON /T/ TOKEN
        COMMON /REFLOC/ REFLOC
        COMMON /DIRECT/ UP, DOWN
        LOGICAL UP, DOWN
        INTEGER S (2)
        LOGICAL OK
        WRITE (3,*) "Enter PLSTST"
C
C A parm list consists of a left paren followed by 1 or
C more expressions separated by commas and terminated by
C a right paren.
```

```
C
      IF (TOKEN .NE. @LEFTP)
           DOWN = .FALSE.; UP = .TRUE.
         (TOKEN .EQ. @LEFTP)
           REFLOC = @PARM
           S (1) = @EXPRESSION; CALL EMPSTR (S (2))
           CALL PUSH (S,2)
      FI
      RETURN
      END

      SUBROUTINE PLSTND (OK)
      IMPLICIT INTEGER (A-Z)
      COMMON /T/ TOKEN
      COMMON /REFLOC/ REFLOC
      COMMON /DIRECT/ UP, DOWN
      COMMON /CURSTR/ CURSTR
      COMMON /CALL/ CALL
      INTEGER S (5), COMMA (1)
      LOGICAL CALL, OK, UP, DOWN, EMPTY, EMPTST
      DATA COMMA /1H,/
      WRITE (3,*) "Enter PLSTND"
C
C As each parameter is completed add its string to the
C parameter list string, separating them with commas.
C if the next token is a comma, then get another parm.
C
      CALL POP (S,LEN)
      EMPTY = EMPTST (S (2))
      IF (EMPTY)
           CALL JOINST (S (2),CURSTR)
         (.NOT. EMPTY)
           CALL NEWSTR (TEMP,COMMA,1)
           CALL JOINST (TEMP,CURSTR)
           CALL JOINST (S (2),TEMP)
      FI
      IF (TOKEN .EQ. @COMMA)
           CALL PUSH (S,2)
           REFLOC = @PARM
           S (1) = @EXPRESSION; CALL EMPSTR (S (2))
           CALL PUSH (S,2)
           DOWN = .TRUE.; UP = .FALSE.
         (TOKEN .NE. @COMMA)
           CURSTR = S (2)
           CALL = .TRUE.
      FI
      RETURN
      END
```

```
      SUBROUTINE SUBST (OK)
      IMPLICIT INTEGER (A-Z)
      COMMON /REFLOC/ REFLOC
      INTEGER S (4), STR (4)
      LOGICAL OK
      WRITE (3,*) "Enter SUBST"
C
C A subscript list is any number of simple expressions,
C separated by commas and enclosed in parentheses.
C If the last structure was a Form, make sure that the
C structure of the Form is an array.
C
      REFLOC = @SUBSCRIPT
      CALL POP (S,LEN)
      LAST = S (2); PTR = S (3)
      IF (LAST .EQ. @ARRAY)
           CALL PUSH (S,4)
           S (1) = @SIMPLEXP; CALL EMPSTR (S (2))
           CALL PUSH (S,2)
         (LAST .EQ. @FORM .OR. LAST .EQ. @CURRENT)
           NEXT = PTR; CALL NEXTLL (NEXT,@DOWN)
           CALL NEXTLL (NEXT,@DOWN)
           CALL NEXTLL (NEXT,@ACROSS); STRPTR = NEXT
           CALL GETLL (STRPTR,STR)
           IF (STR (1) .EQ. @ARRAY)
                S (2) = @ARRAY; S (3) = STRPTR
                CALL PUSH (S,4)
                S (1) = @SIMPLEXP; CALL EMPSTR (S (2))
                CALL PUSH (S,2)
              (STR (1) .NE. @ARRAY) OK = .FALSE.
           FI
      FI
      RETURN
      END

      SUBROUTINE SUBND (OK)
      IMPLICIT INTEGER (A-Z)
      COMMON /T/ TOKEN
      COMMON /REFLOC/ REFLOC
      COMMON /DIRECT/ UP, DOWN
      COMMON /CURSTR/ CURSTR
      INTEGER TAIL (2), DIGIT (10), LOCAL (6), TYPEOF (5), DOPE (4)
      INTEGER ARR (4), S (4), RNG (2)
      LOGICAL OK, UP, DOWN
      DATA TAIL /")","*"/
      DATA DIGIT /"0","1","2","3","4","5","6","7","8","9"/
      DATA LEFT /"("/, RIGHT /")"/, COMMA /","/
      DATA LOCAL /"L","O","C","A","L","("/
      DATA TYPEOF /"T","Y","P","E","("/
```

```
      DATA DOPE /"D","O","P","E"/
      DATA PLUS /"+"/
      WRITE (3,*) "Enter SUBND"
C
C If the next token is a comma, then there are more
C subscripts to follow.  Add the expression string to
C the subscript list string.
C
      IF (TOKEN .EQ. @COMMA)
           CALL POP (S,LEN)
           CALL JOINST (S (4),CURSTR)
           CALL ADDCHR (S (4),COMMA)
           CALL PUSH (S,4)
           REFLOC = @SUBSCRIPT
           S (1) = @SIMPLEXP; CALL EMPSTR (S (2))
           CALL PUSH (S,2)
           DOWN = .TRUE.; UP = .FALSE.
C
C If it is not a comma, then the subscript list is complete.
C Generate an access expression as follows
C           DOPEn(rangelist,parmlist)*basesize
C n is the number of ranges in this array.  The rangelist is
C the upper and lower bounds for this array.  If this array
C is part of the current form, then the rangelist will be
C in the dope vector TYPE.  If not, then it will be in the
C dope vector LOCAL.  The base size is in the symbol table
C entry for this array.  If it is known then generate the
C value, if not generate TYPE(addr).
C
         (TOKEN .NE. @COMMA)
           CALL POP (S,LEN)
           ARRPTR = S (3)
           STRPTR = ARRPTR; CALL NEXTLL (STRPTR,@ACROSS)
           CALL TYPREF (STRPTR)
           CALL GETLL (ARRPTR,ARR)
           NUMRNG = ARR (4)
           CALL NEWSTR (TEMP,DOPE,4)
           CALL ADDCHR (TEMP,DIGIT (NUMRNG+1))
           CALL ADDCHR (TEMP,LEFT)
           RNGPTR = ARRPTR; CALL NEXTLL (RNGPTR,@DOWN)
           CALL GETLL (RNGPTR,RNG)
           IF (RNG (1) .EQ. @KNOWN) CALL NEWSTR (TEMP2,LOCAL,6)
              (RNG (1) .EQ. @UNKNOWN) CALL NEWSTR (TEMP2,TYPEOF,5)
           FI
           CALL JOINST (TEMP,TEMP2)
           ADDR = RNG (2)
           CALL NUMSTR (TEMP2,ADDR)
           CALL ADDCHR (TEMP2,RIGHT)
           CALL ADDCHR (TEMP2,COMMA)
```

```
            CALL JOINST (TEMP,TEMP2)
            CALL JOINST (TEMP,CURSTR)
            CURSTR = TEMP
            CALL NEWSTR (TEMP,TAIL,2)
            CALL JOINST (CURSTR,TEMP)
            IF (ARR (2) .EQ. @KNOWN)
                 BASE = ARR (3)
                 CALL NUMSTR (TEMP,BASE)
                 CALL ADDCHR (TEMP,PLUS)
                 CALL JOINST (CURSTR,TEMP)
               (ARR (2) .EQ. @UNKNOWN)
                 CALL NEWSTR (TEMP,TYPEOF,5)
                 ADDR = ARR (3)
                 CALL NUMSTR (TEMP2,ADDR)
                 CALL ADDCHR (TEMP2,RIGHT)
                 CALL ADDCHR (TEMP2,PLUS)
                 CALL JOINST (TEMP,TEMP2)
                 CALL JOINST (CURSTR,TEMP)
            FI
       FI
       RETURN
       END

       SUBROUTINE WRAP
       IMPLICIT INTEGER (A-Z)
       COMMON /REFLOC/ REFLOC
       COMMON /REFTYP/ REFTYP, TYPSET
       COMMON /CURSTR/ CURSTR
       COMMON /T/ TOKEN
       LOGICAL TYPSET
       INTEGER CVTR (5), CVTL (5), RP (1)
       DATA CVTR /"C","V","T","R","("/
       DATA CVTL /"C","V","T","L","("/
       DATA RP /")"/
C
C All form type variables are generated as vectors of integers.
C Any non-scalar variables or components can only be operated
C on by subroutines or functions.  However Fortran scalars may
C be operated on by infix arithmetic.  This causes a problem
C since a reference to a real or logical component will be
C generated as a reference to an integer array.  In order to
C correct this, a type changing function is wrapped around
C the reference.
C
       WRITE (3,*) "In WRAP"
       IF
         (REFLOC .EQ. @LHS)
         (REFLOC .NE. @LHS)
           IF
```

```
            (.NOT. TYPSET)
            (TYPSET)
              TYPSET = .FALSE.
              IF
                (REFTYP .EQ. @INTEGER)
                (REFTYP .NE. @INTEGER)
                  IF
                    (REFLOC .EQ. @PARM .AND. (TOKEN .EQ. @COMMA .OR.
     X                                        TOKEN .EQ. @RIGHTP))
                    (REFLOC .NE. @PARM .OR. (TOKEN .NE. @COMMA .AND.
     X                                       TOKEN .NE. @RIGHTP))
                      WRITE (3,*) "Wrapping"
                      IF
                        (REFLOC .NE. @PARM)
                        (REFLOC .EQ. @PARM)
                          REFLOC = @RHS
                      FI
                      IF
                        (REFTYP .EQ. @REAL)
                          CALL NEWSTR (TEMP,CVTR,5)
                        (REFTYP .EQ. @LOGICAL)
                          CALL NEWSTR (TEMP,CVTL,5)
                      FI
                      CALL JOINST (TEMP,CURSTR)
                      CURSTR = TEMP
                      CALL NEWSTR (TEMP,RP,1)
                      CALL JOINST (CURSTR,TEMP)
                  FI
              FI
          FI
        FI
        RETURN
        END

        SUBROUTINE TYPREF (PTR)
        IMPLICIT INTEGER (A-Z)
        COMMON /REFTYP/ REFTYP, TYPSET
        LOGICAL TYPSET
        INTEGER STR (4), SPC (3)
C
C This routine records the type of a reference if it is
C a Fortran scalar or a form which is a Fortran scalar.
C
        WRITE (3,*) "In TYPREF"
        STRPTR = PTR
        CALL GETLL (STRPTR,STR)
        IF (STR (1) .NE. @FORM)
           (STR (1) .EQ. @FORM)
             LOOP
```

```
        WHILE (STR (1) .EQ. @FORM)
        LOOPBY
          CALL NEXTLL (STRPTR,@DOWN)
          CALL GETLL (STRPTR,SPC)
          STRPTR = SPC (1)
          CALL NEXTLL (STRPTR,@DOWN)
          CALL NEXTLL (STRPTR,@DOWN)
          CALL NEXTLL (STRPTR,@ACROSS)
          CALL GETLL (STRPTR,STR)
      ENDLOOP
FI
IF (STR (1) .NE. @FORTRAN)
   (STR (1) .EQ. @FORTRAN)
     TYPSET = .TRUE.
     REFTYP = STR (4)
FI
RETURN
END
```