COMPUTER SYSTEMS LABORATORY

STANFORD UNIVERSITY STANFORD CA 943054055

UFORT

A Fortran-to-Universal-Pcode Translator

(FIXFOR-2)

CSL Technical Report No. 168

Stanford University

January 30, 1980
13:06

Frederick Chow
Peter Nye
Cio Wiederhoid

TABLE OF CONTENTS

Section Page

TABLE OF CONTENTS

Acknowledgement

This work was performed as part of the software development' effort at Stanford University, under a subcontract from Lawrence Livermore Laboratory to the Computer Science Department, Principal Investigator Professor Gio Wiederhold, Contract No. LLL PO 9083403. The S-l hardware and software development has been supported by the Department of the Navy via Office of Naval Research Order Numbers N00014-76-F-0023, N00014-77-F-0023, and N00014-78-F-0023 to the University of California Lawrence Livermore Laboratory (which is operated for the U. S. Department of Energy under Contract No. W-7405-Eng-48), from the Computations Group of the Stanford Linear Accelerator Center (supported by the U. S. Department of Energy under Contract No. EY-76-C-03-05 1 5), and from the Stanford Artificial Intelligence Laboratory (which receives support from the Defense Advanced Research Projects Agency and the National Science Foundation).

The UFORT compiler is a direct derivative of the earlier PCFORT compiler written by Fernando Castaneda, Frederick Chow, Peter Nye, Daniel Sleator and Gio Wiederhold [CCN79]. The changes were implemented by Frederick Chow, who also assumes responsibility for this document. We are no longer maintaining the PCFORT version.

We also wouid like to acknowledge the invaluable assistance of Erik Gilbert, Curt Widdoes, and David Fuchs during the course of the development from PCFORT to UFORT.

1. Introduction

The Fortran compiler described in this document, UFORT, was written specifically to serve in a Pascal environment [JeW78], using the Universal P-Code as an intermediate pseudo-machine [NA J75]. The need for implementation of Fortran these days is due to the great volume of existing Fortran programs, rather than to a desire to have this language available to develop new programs. We have hence implemented the full, but traditional Fortran standard [ANS64, ANS66], rather than the recently adopted augmented Fortran standard [ANS76]. All aspects of Fortran which are commonly used in large scientific programs are available, including such features as SUBROUTINES, labelled COMMON, and COMPLEX arithmetic. In addition, a few common extensions, such as integers of different lengths and assignment of strings to variables, have been added.

1.1 Objectives and constraints

The foremost objective in the design of this compiler is the generation of correct code. Effects of this objective are a clean approach to the design of the compiler, the use of Pascal as the implementation language, and the use of a simple one-pass compiling technique. The one-pass approach has led to two additionai constraints on the source language: variable declarations, if given, must precede all executable statements within each program unit, and keywords must be separated from variable identifiers by blanks. These constraints are commonly followed by programmers, but are not part of the standard. A pass over Fortran source code with a text editor can easily correct failures to obey that constraint, since these changes do not affect the semantics of Fortran programs in any way. We feel of course that such constraints are a reasonable part of any programming environment we wish to support. UFORT does not depend on reserved words in its method to recognize keywords and is hence extensible to additional statement types. Candidates for additions are several file manipulation statements, now used by existing compilers and defined in [ANS76], and other features to support real-time operations and aspects of parallel processing.

The structure of the compiler is derived from a Fortran compiler, written in Fortran, which was used for student programming from 1963 to 1967 at UC Berkeley (Student) on an IBM 7094 *system.* A derivative of that compiler is the PL/ACME compiler [BRW68], a compiler for a subset of PL/1, also written in Fortran, with strong support for on-line laboratory operations. Writing the new compiler in Pascal has allowed formalization of modular concepts used in the earlier compiler [WiB70]. The availability of recursion has caused us to switch to the use of recursive descent as the method for compiling arithmetic instructions, a method which copes well with some of the problems of Fortran syntax.

The compiler, while attempting to generate good U-Code, does no explicit optimization of the generated code. Recognition of common subexpressions, for instance, wrli require at least an additional pass in a compiler. Current research in the Pascal/P-Code project at UCSD is leading to such an optimizer operating on U-Code [SPT79]. The compiler also makes only very general assumptions about the register structure in the underlying machine. It is the function of a U-Code compiler (e.g. SOPA [Zel80]) or a U-Code interpreter (e.g. UASMINT [Bsh79]) to carry out the requested U-machine actions in a manner which utilizes the underlying hardware effectively.

The original P-Code generated is a direct derivative from the original work of associates of N. Wirth at the ETH [NAJ75], and documented by us in an S-1 project documentation note

[GiW77]. This was later adapted to the Universal P-Code defined by the UCSD Optimizer Project [PSi79]. In our case the U-Code is compiled into machine-code for the S-l processor [FiZ78], a very high speed machine with a 36-bit-word architecture, which also supports 72-bit double-word, 18-bit half-word, and 9-bit quarter-word or byte operations. We hence expect 4 bytes per word; that is 360-style alphabetic variables. This aspect does not affect the UFORT compiler itself, but is of major concern when transporting Fortran application programs, which manipulate characters, between computers, since Fortran standards have ignored the issue of character-to-word relationships.

The associated run-time package is of course sensitive to the machine architecture. The dependencies are easy to manage since this package is written in Pascal. The U-Code generated from the run-time by our Pascal compiler can be combined with the U-Code from UFORT before being interpreted, or the run-time U-Code can be translated to machine code and loaded for execution together with the machine code translated from Fortran programs via UFORT. The run-time package is hence easily changed or augmented by more Pascal-written routines. This approach also makes available to Pascal programs the FORMAT conversion routines implemented within the Fortran run-time package.

The two components which make up UFORT, the compiler and the run-time package, are of course constrained due to the facilities provided by the Pascal U-Code environment. The most serious of these is no doubt the unavailability of direct access to files. We plan to extend our system with direct files supporting variable length records, and at that time both Fortran and Pascal will be augmented to support these features [AKe80].

Another aspect of the U-Code environment is that it does not sufficiently provide for the separate compilation of routines. UFORT wiil hence accept a complete set of program units (the main program, any BLOCK DATA program, all SUBROUTINEs and FUNCTIONs together) and generate a single block of executable U-Code. After translation to S-l machine code the resulting relocatable instructions can be combined with other program units through the use of a linking loader [KeW 791.

1.2 Conclusion

The UFORT Fortran compiler is a building block within a Pascal and U-Code environment, which can take care of existing needs for the continued use of Fortran coded algorithms. By bringing Fortran into this environment, a dichotomy of programming approaches can be avoided, and a more consistent approach to computing can result.

The next section specifies the Fortran source statements recognized by UFORT, together with the differences from the standard. The remainder of this document describes the implementation in sufficient detail to serve ongoing maintenance and extension needs.

2. User's Guide

This section describes the limitations and extensions of UFORT Fortran in comparison with standard Fortran compilers, and especially in comparision with the full Fortran '66 Standard [ANS66].

2.1 Statements

The following Fortran statement types have been Implemented:

Declaration statements:

DIMENSION
COMMON
EQUIVALENCE
IMPLICIT
EXTERNAL
LOGICAL
INTEGER
COMPLEX
REAL
DOUBLE PRECISION
DATA

Executable statements:

The assignment statement
ASSIGN
IF (logical and arithmetic)
GOTO (unconditional, computed, and assigned)
CALL
RETURN
PRINT
STOP
DO
READ
WRITE
REWIND or OPEN

Other statements:

The statement function declaration
FORMAT
FUNCTION
SUBROUTINE
BLOCK DATA
SET
CONTINUE
END
ENTRY

Not implemented:

END FILE
BACKSPACE
PAUSE

2.2 Program format

Some restrictions on program format are imposed by UFORT:

Source text format:

Identifiers, including keywords, must be separated by delimiters. For example, "DO30I =1, 3" is illegal; it should be "DO 30 I =1, 3". Similarly, "COMMONA,B" should be "COMMON A,B". Blanks are not allowed within identifiers, keywords and real constants. Blanks within dotted keywords, however, are allowed (e.g. ". TR U E. ").

The usual convention of specifying a quote embedded within a quoted literal using two consecutive quotes is followed.

Blank lines are allowed. A line cannot contain more than one statement.

Position of declaration statements:

All declaration statements, including DATA statements, must appear before the first executable statement in a program unit. Statement functions must appear after the declarative statements and before the first executable statement. The only restriction regarding the order among the declaration statements is that the type and dimension declaration of a variable must precede its initialization specification.

FORMAT statements may appear either with the declarative or the executable statements.

if an IMPLICIT statement is used, it must be the very first statement in the program unit.

Variable names:

Fortran keywords and standard and intrinsic function names can be used as variable names, except the keyword FORMAT. Also, the name of a common block or an ENTRY statement in the same program unit can be the same as a variabie name. However, the same name cannot be used in a single program unit as both a variable name and a standard, intrinsic, or user-defined subprogram name. If a name is longer than 6 characters, the extra characters are ignored and a warning is given.

FORMAT specifications:

Commas are not mandatory in FORMAT specifications if they cause no ambiguity, For example,

```
        (X3XX'ONE'X/X2(4HFOURF8.5I6))

and     (X,3X,X,'ONE',X,/,X,2(4HFOUR,F8.5,I6))
```

are equivalent.

If a FORMAT specification is to be kept in an array, any embedded quote that occurs in the FORMAT has to be replicated when stored in the array. Another level of replication might be required in specifying the quote in the program text. In the following example, the FORMAT co output the word "DON'T" is stored in the array FSTR:

```
INTEGER FSTR*2(5) /'(''DON''''T'')'/
```

Statement labels:

Only executable statements and FORMAT statements can be assigned labels.

2.3 Data types and constants

2.3.1 Data types

Variables and functions may be of type INTEGER, REAL, COMPLEX, or LOGICAL. The standard naming conventions are used to determine if a variable or function is of type integer or real (names starting with letters from I to N denoting integers), but they may also be explicitly declared. The naming conventions may also be overridden through the use of an IMPLICIT statement.

The following precisions are possible:

LOGICAL: quarter word, half word, single word (default) and double word;

INTEGER: quarter word, half word, single word (default) and double word;

REAL: half word, single word (default) and double word;

COMPLEX: two single words (default) and two double words.

Precisions are specified in quarter words, as in IBM Fortran:

```
INTEGER*1AAA
LOGICAL*8 BBB
COMPLEX CCC
COMPLEX*16 FUNCTION DDD
DOUBLE PRECISION EEE
REAL*8 EEE
```

Automatic conversion occurs whenever necessary between and among any precisions of the integer, real and complex types. Real numbers are converted to integers by truncation. Conversion to complex number is done by adding a zero imaginary part. When a complex number is converted to real or integer, its imaginary part is discarded.

Integer variables used as the control variable of a DO statement, for storing a label or for storing a device number for use in a READ, WRITE or REWIND statement must be of single precision.

2.3.2 Constants

The upper limits allowed for integer values are 255 for quarter-word integers, 13 1071 for half-word integers, 34359738367 for full-word integers and 73786976294838206463 for double-word integers. The lower limits are 1 less then the negatives of these numbers. The upper and lower limits for reals are I.70141 1843E+38 and 1.469368010E-39 respectively, for all precisions.

Complex numbers consist of a left parenthesis, a real expression, a comma, another real expression, and a right parenthesis. Thus (.3*X,SIN(Y)) is a legal complex number.

2.4 Arrays and storage management

Array subscripts:

Array subscripts may consist of any legal integer expression. Up to seven dimensions are allowed.

Bound checking for array subscripts, if turned on, is done separately for the subscript of each dimension.

Array bound checking at compile time is done for arrays that appear in COMMON and EQUIVALENCE declarations, and for the ones that are initialized. These arrays cannot have ad justable dimensions.

The specification of array elements in DATA and EQUIVALENCE statements with only one dimension for arrays of several dimensions is accepted. For example, for an array dimensioned as A (3,3), the array element A(2,3) may be specified as A(8).

Arrays with adjustable dimension:

No restriction is made on the value of an actual argument that represents the bound of an array in the argument list of a subprogram. I.e. no check is made that the value is within the declared bound of the actual array parameter. When an array subscript is beyond the range of the actual array, no assumption should be made as to the referenced value. (The same applies in the case of arrays with constant bounds when the bound declared for the actual array parameter differs from that declared for the formal array parameter, or when their dimensions are different.)

In the subprogram, bound checking (if turned on) for an array with adjustable dimension is made against the current value of the argument used in the dimension declaration. Change to the value of this dummy argument is allowed in the subprogram. If the actual argument is an uninitialized integer variable, no assumption should be made as to the declared bound in the subprogram.

COMMON declarations:

There are two special areas which are used for the common variables, one is used for the blank common area and the other is for the rest of the common areas. The blank common may be of any different length in each program unit, as specified in [ANS76]. The COMMON declaration of any labelled common may not require a storage area larger than the amount specified in the first declaration of the common, as in the following example:

```
wrong:                        right:

    COMMON /X/ A                  COMMON /X/ A,DUMMY
    DIMENSION A(20)               DIMENSION A (20), DUMMY (10)
    END                           END

    SUBROUTINE R                  SUBROUTINE R
    COMMON /X/ B                  COMMON /X/ B
    DIMENSION B(30)               DIMENSION B(30)
    END                           END
```

Alternatively, it is possible to use the CSIZ switch that fixes a minimum size for the common areas. If the length occupied by a common area in its first declaration is smaller than that specified in any of its later declarations, the switch should be set to the space needed for the larger one.

Storage allocation:

No assumption should be made about the location of one variable or array in relation to another outside a common area.

Additional quarter-words are inserted as necessary to align half-words on half-word boundaries, single-words on single-word boundaries and double-words on double-word boundaries. Thus, a quarter-word variable followed by a single-word variable in a common area would require two full words of storage.

2.5 Initializing variables

Variables can be initialized in both DATA and type declaration statements. The type declaration statement with initiaiitations and DATA statement are formed as follows:

$\text{type}*s\ a*s_1(k_1)/x_1/, b*s_2(k_2)/x_2/, \ldots, z*s_3(k_3)/x_3/$

$\text{DATA}\ a(k_1), \ldots, d(k_4)/x_1/, e(k_5), \ldots, h(k_8)/x_2/, \ldots$

where

type is INTEGER, REAL, LOGICAL, DOUBLE PRECISION or COMPLEX;

$*s_1, *s_2, \ldots$ are optional, each s representing one of the permissible length specifications for its associated type;

a, b, ..., z are variable or array names;

$(k_1), (k_2), \ldots$ give dimension information for arrays in declaration statements and subscript information for array elements in DATA statements. In a declaration statement, this always specifies the entire array. If absent for an array in a DATA statement, short form specification for the entire array is impiied;

$x_1, x_2, \ldots$ are constants or lists of constants. $/x_1/, /x_2/, /x_3/ \ldots$ are optional in a declarative statement, and are used to specify initial values for single preceding variables and array names. In a DATA' statement, they are not optional, and specify initial values for the preceding list of variables, array elements or array **names;**

2.5.1 Loops in variable lists in DATA statements

Nested loops are allowed in specifying variable lists in a **DATA** statements. The form of these loops is similar to that used in the **READ** and **WRITE statements.** Syntactically, each variable or array element **in** &he above specification "(a(k_1)", e.g., can be replaced by **a** pair of parantheses enclosing a list of variables or array elements. The loops can be nested to any arbitrary depth. The general form of the loops is:

DATA ((..., $i=l_1,m_1,n_1$), (. . . , $j=l_2,m_2,n_2$), . . . , $k=l_3,m_3,n_3$), . . . /x_1/, . . .

where

i, j, k are control variables. Their appearances imply that they can be used in specifying subscripts among the array elements which occur anywhere inside the loop. The control variables have no relation to any other regular variable with the same **name in** the program, and **they** do not obey the implicit typing since they must be integers. If **a** control variable name occurs more than once in a single nesting of loops, the **one** in the level nearest its occurrence in a subscript is effective when the subscript is **inside** the ranges of both loops;

l_1, l_2, . . . , ml, m_2, . . . **and** n_1, n_2, . . . specify lower bounds, upper bounds and step amounts respectively for the loops. The appearance of the step amount is optional.

2.5.2 General initialization rules

1. The type of **initialization** is determined by the type of the constant specified, and not by the **type** of the variable being initialized. Only the size of the variable affects **the initialization.**

2. The initialization of arrays is done in storage order. In **a** declarative statement, each list of constants must correspond in number to the preceding variable or array. In a DATA statement, the correspondence is to the total number of variables and array elements specified in the preceding list, taking account of loop iterations if any. If extra constants are **given,** they are ignored. If not enough constants are given, the extra **variables** or array elements are **not** initialized. In **both** cases, warnings are **given.** A **complex** variabie is taken as two reai variables, and they correspond to two initialization constants. The parentheses in specifying a complex constant **are** optional.

3. A replication factor can be used to specify how many times the constant following the asterisk is to be repeated in the initializing process. The **syntax is:**

<rep>*<val>

where <rep> is the replication factor and <val> is the constant value. (E.g. 5*3.2 means that the constant vaiue 3.2 is going to be used 5 times.)

4. Function names or subprogram parameters cannot be initialized.

5. Arrays must be dimensioned before initialization in a **DATA** statement or in a type

declaration statement. Also, any type declaration for a variable in a DATA statement must appear before the DATA statement.

6. If the initializatron of a variable or location is specified more than once, only the last initialization is effective.

2.5.3 Initialization by character strings

The initialization of variables by character strings, in DATA statements or type declaration statements, follows these rules:

1. One character will be stored per quarter-word. A full word has hence the capacity to hold four characters, half- and double-words hold 2 and 8 characters'respectively. An array has a capacity which is the product of its size and the capacity of its elements.

2. If the string is larger than the capacity of the variable being initialized, only the initial characters of the string are used and the rest are discarded.

3. If the number of characters in the string is smaller than the capacity of the variable then the string is padded with NULL (binary zeroes).

4. Character strings may be preceded by a replication factor, followed by an asterisk. The replication factor increases the number of string elements, not their length.

5. An array, or the two halves of a complex variable, may be filled with successive characters from the string. If an element is incomplete, it will be filled with NULL. If successive elements are not reached they remain uninitialized.

Characters can also be assigned to variables using an assignment statement.

2.5.3.1 Examples

Initialization statement:

```
INTEGER M/'ABCD'/, A(2)/'ABCDEFGH'/
DIMENSION C(3), D(3), E(8), F(3)
DATA D(2),D(3),C/'AB','CD','ABCDEFGHI'/
DATA E/'ONEISMORE','TWO','THREE','FOUR','FIVE','SIX','SEVEN'/
DATA F/3*'MOM'/
```

Initializations performed:

VARIABLE	VALUE
M	'ABCD'
A(1)	'ABCD'
A(2)	'EFGH'
D(1)	unintialized
D(2)	'AB'
D(3)	'CD'
C(1)	'ABCD'
C(2)	'EFGH'
C(3)	'I'

```
E(1)      'ONE1'
E(2)      'TWO'      ;earlier, E(2) contained 'SMORE' but this was
                      overwritten with the next element in the list
E(3)      'THRE'
E(4)      'FOUR'
E(5)      'FIVE'
E(6)      'SIX'
E(7)      'SEVE'
E(8)      'N'        ;no more elements in list, thus not overwritten
F(1)      'MOM'
F(2)      'MOM'
F(3)      'MOM'
```

2.6 Subprograms

The restrictions with regard to subprograms are:

Functions:

A statement function must have at least one argument. A function with no parameter must be declared EXTERNAL in each program unit in which it is referenced. Otherwise, the function name is taken as a variable name.

Parameters to Subprograms:

All parameters are passed by reference, including array elements used as arguments. Thus their values can be altered as the result of a subprogram call.

External Subprograms:

Currently, all program units used in a program are compiled at the same time as the main program; separately compiled subroutines or functions have not yet been Implemented.

2.7 Subprogram names as parameters

Subprogram names can be passed as parameters in a call to another subprogram, and they can be passed onwards in another call in the subprogram to which they have been passed. If a subprogram name (or a parameter representing a subprogram name) to be passed as parameter has not been called explicitly previously in that program unit, it must have been declared EXTERNAL. This rule is for ensuring that the compiler can diagnose that the actual parameter is a subprogram name.

A format parameter representing a subprogram name cannot be used also as a variable inside a program unit. However, the same parameter can be used to represent more than one functions or subroutines in different calls, and they can have varying number of call parameters.

Statement functions cannot be passed as parameters, but a statement function can have subprogram name parameters.

2.8 Multiple entries to subprograms

Multiple entry subprograms, though not part of the standards, are supported in the way they are usually used. The keyword ENTRY has to be placed as the last symbol in the regular subprogram heading (SUBROUTINE or FUNCTION statement) to indicate the presence of ENTRY statements in the subprogram.

The ENTRY statements, which indicate possible entry points to the subprogram, must only be in the executable part of the subprogram. An ENTRY name has no connection with any other possible identical local name in that program unit. An ENTRY statement is regarded as the declaration of a new program unit to the test of the Fortran program, and they can be called there as if they were unique program units. If the ENTRY statement belongs to a function subprogram, the ENTRY name is automatically made a function of the same type. The ENTRY name must not be typed explicitly in any way, even if its type is not the same as that implied by its name.

Parameters can be used for ENTRY statements. Any ENTRY parameter used must appear the first time as a parameter either in the subprogram heading or an ENTRY statement, except that it can possibly be typed or declared EXTERNAL in the declaration part of the program unit.

An ENTRY statement has no effect on the normal flow of control in the program unit if it is not called directly.

In the following example of a multiple entry subroutine, a call to the subroutine SETVAL determines the variable whose value is to be used in assignment in any subsequent call to the en try ASSIGN:

```
SUBROUTINE SETVAL (P1) ENTRY
RETURN
ENTRY ASSIGN(P2)
P2 = P1
RETURN
END
```

2.9 User options: the SET statement

Options are specified using the SET statement. Option names are identified by the first 4 letters only. More than 1 option can be specified in a SET statement by using commas. E.g. "SET GENC = T, ASTR = F". T turns options on, and F turns them off.

Here are the options implemented in UFORT. Options related to U-Code translators or interpretors are not included here:

BCHK - When T, execution time bound checking on array subscripts is turned on. Defau lt is F.

GENComment - When F, no U-Code comment is written on the U-Code file. Default is T. The LOC instruction in U-Code is regarded as comment in this case.

CSIZ - The argument is a number. It specifies the minimum size in number of words to be allocated to the common areas that appear for the first time in the next COMMON statement. It is reset to 0 at the end of each COMMON statement and at the beginning of each program unit.

TPRM - The argument is a number. It gives how many parameters should be passed in registers. Default is 10. Maximum is 15.

2.10 *Input/Output*

2.10.1 File handling

UFORT uses Pascal run-time routines for input and output on the character level.

Pascal treats all I/O as being to files of characters. Fortran device numbers 0 through 5 are given internal representations of FILE0, FILE1, FILE2, ... FILES. Provisions exist for extending the number of devices to above 5. The mapping between these pseudo-files and actual devices or disk files is done at execution time, usually by a direct prompt at the terminal. E.g.

```
FILE1? DATA1
FILE23 OUT1
FILE3? TTY:
```

A file is opened immediately after the prompt is answered. This may occur at the beginning of the program or at the first appearence of a READ or WRITE statement using the device number of the file, depending on the Pascal run-time used. (For the S-l, these are specified in [GWa78] and the current [HiN80].) Files are always closed only at the end of the program.

Random access within files is not allowed; files must be written to or read from starting at the beginning of the file. The first time in a program a file is written to, its previous contents are destroyed, and the file pointer is reset to point to the beginning of the file. A file may be both read from and written to in the same program, but each successive change of mode causes the file pointer to be reset to point to the beginning of the fiie. The file pointer may be explicitly reset to point to the beginning of the file with the Fortran statement REWIND. In the current run-time, a change of mode or a REWIND will also cause another prompt for the name of the file. OPEN is an alternative name for REWIND.

The BACKSPACE and END FILE statements are not implemented.

2.10.2 The READ and WRITE statements

The standard READ, WRITE and FORMAT statements use Fortran run-time routines. Both formatted and unformatted reads and writes are handled. Unformatted writes use fields of fixed widths according to the types of the variables being output. In unformatted input, the input file is always scanned until the next non-blank character in the input file is found. Blanks are taken as delimiters, and they do not have to be present if there is no ambiguity. Comma should not be used as delimiters. Each unformatted READ or WRITE statement starts on the next line.

The maximum length of an input or output line is 256 characters. Any output to beyond the 256th character will automatically cause an extra new line to be written. An input line longer than 256 characters is processed as a single line but anything beyond the 256th character is treated as blanks. If an input line is shorter than that specified in the format specification, an error message is given.

Any internally representable character can be output via an A-formatted field. The writing of control characters like the carriage-return or line-feed to an A-formatted field may cause the form of the output line to depart from that specified in the format specification.

The execution error messages of the READ and WRITE statements go to file OUTPUT.

2.10.3 The PRINT statement

Apart from the READ, WRITE and FORMAT statements, the PRINT statement, which makes use of Pascal run-time routines, and acts somewhat like a Pascal WRITE statement, allows the bypassing of the Fortran run-times in performing output operations. It prints integers, reals, booleans, string constants, or complex numbers, or any legal expressions containing these items.

Normally, a carriage-return line-feed will be printed at the end of the line. This may be suppressed by adding a semicolon.

A field width may be added to any item. This indicates the maximum length of the item to be printed. Enough blanks will be added to make the item always have that length. The default field widths are 14 for integers and reals, and the actual length of the string for strings.

Output always goes to the Pascal standard file OUTPUT.

Here are some examples:

```
PRINT 'THE ANSWER IS', X*2                 result: THE ANSWER IS 4.0

PRINT 'THE ANSWER IS';                     result: THE ANSWER IS 4.0
PAINT X*2

PRINT 'THE ANSWER IS':20,X*2:10            result: THE ANSWER IS          4.0

COMPLEX*8 x                                result: THE ANSWER IS 2.0       8.0
PRINT 'THE ANSWER IS', X*(2.,0.):10

PRINT:2 'THE ANSWER IS',X*2                result: THE ANSWER IS 4.0
```

2.11 *Miscellaneous*

<u>DO statement:</u>

An integer expression may be used as the lower bound, upper bound or step amount. The control variable must not be an array element. The default step size is 1. Negative step sizes are allowed.

In the case that the upper bound or step size is an integer variable, if a change is made to the value of the variable during execution of the loop, the upper bound or step size is changed accordingly.

Jumping into the range of a DO loop (including the terminal statement) from outside the DO range is allowed. The control variable assumes the value it has at the time of the jump. If the control variable is not initialized, no assumption should be made as to the value of the variable.

A DO loop cannot be closed by a FORMAT statement.

Use of integer variables as label variables:

No distinction is made between integer variables and label variables. Le. the usage of an integer variable is not restricted with regard to whether it has assumed its value by regular integer assignments or by the ASSIGN statement for statement labels. An array element can be used for the variable.

Bitwise operations on variables:

The bitwise .AND., .OR. and .NOT. operations on integer, real and complex values are allowed. The operands are checked for type compatibility as in the case of other arithmetic operations.

Intrinsic and standard functions:

When the intrinsic and standard functions are used, their types are not affected by implicit or explicit typings.

3. Overall Organization

3.1 Structural scheme

UFORT's processing of an input user program is driven by its main procedure and procedure BLOCK, which invoke the various modules either directly or indirectly. The organization of UFORT is based on these modules. It is structured according to the relationships among the various modules. Despite its length (about 9000 lines), UFORT is easily understood once its structure is revealed.

When the compiler processes a given program statement, it either generates code from it or remembers the information given in the program text by building some internal structure, which invariably is a linked list of a particular type. A module in UFORT satisfies at least one of the following conditions:

1. It scans and processes a type of statement in the user program.

2. It scans and processes a specific construct which occurs in more than one type of statement. These are:
   (a) the arithmetic expression processor,
   (b) the procedures for loading and storing variables,
   (c) the procedure to process function calls,
   (d) the procedures to process initialization specifications.

3. It processes an internal structure, and possibly generates code from it. These are:
   (a) the procedure to close either a DO loop or a loop in an I/O statement,
   (b) the storage allocation procedure,
   (c) the variable initialization code-generating procedure.
   (d) the procedures to generate code related to multiple entry procedures.

4. It manages an internal table:
   (a) the symbol table routines,
   (b) the standard function table routines,
   (c) the temporary storage management routines.

5. It is a pre-processing procedure for each input statement:
   (a) the lexer,
   (b) the statement classifier.

Apart from these are the error and warning routines, the code-generating routines, the type-checking routines and a number of general utility procedures. Some of these utilities scan and process specific constructs:

(a) procedure GETHTYPE - processes an explicit type specification. E.g. LOGI CAL.
(b) procedure GETTYPE - processes the "*" modification of a type specification. E.g. "* 4".
(c) procedure GETCOORDINATE - processes the subscript specification of an array element in a DATA or EQUIVALENCE statement. E.g. "A(1,3)".
(d) procedure ISARRAY - processes the dimension specification in the declaration of an array, which occurs in the DIMENSION, COMMON and type declaration statements. E.g. "B(1,4)".

3.2 Error handling

UFORT always checks the validity of a program construct before it operates on it. In this way, it safeguards itself from execution errors during compilation. It distinguishes between two kinds of errors:

1. Errors discovered while scanning a program statement: UFORT will stop processing the statement at the point where the error is discovered. The error message is output with '?' printed under the word that causes the error. At most one error message will thus be output for a single statement. In some cases, UFORT will try to generate extra dummy U-Code to make the code already generated for the statement acceptable by the U-Code translator. UFORT will continue to parse and generate code for the rest of the statements in the user program.

2. Errors discovered while processing an internal structure of the compiler: For this type of error (called SPECIAL_ERROR in the compiler), the error message is printed with a name that tells from where the error originates. The recovery procedure may involve deleting the trouble-causing element or altering its contents to make it compatible with the rest of the program. Such actions are invisible to the user.

To enable the features of 1, the statement processing procedures in the compiler always use the global lexeme pointer LXC as index while scanning a statement. The error routine will print '?' under the word that LXC points to. Since different parts of a statement are usually processed by different procedures, the unifying rule used is that each procedure is entered with LXC pointing to the firs& lexeme it processes and exits with LXC pointing to the one after the last lexeme it processes.

Warnings are output when errors are discovered in the program which UFORT thinks will not drastically affect the normal execution of the rest of the user program. Regardless of when it is discovered, only a name will be printed with the message. The position where the warning is printed in relation to the program statements in the listing file serves as another clue to the user in some cases. Recovery actions may also be taken by UFORT. The resulting behaviour of the program is easily predictable by the user.

UFORT always prefers warning instances to error instances. I.e. for each user error, UFORT classifies it as an error instance only if it cannot make it a warning instance.

4. Lexer

4.1 Summary

The purpose of the lexer is to split the input program up into nice pieces, *lexemes*, which are easier to deal with than characters.

Each time the lexer is called it reads the next Fortran statement from the source file. moves it **character** by character into an array called LEXSTRING, stores the Fortran statement label in LABNO, generates the sequence of lexemes contained in this statement, and puts the lexemes into an array called LEXEME. Comments are skipped, and all lines of the source file are copied to the listing file. The length of the string is stored in LEXSTRLENGTH, **the** number of **lexemes in** LEXCOUNT, **the** number associated to **the** first line of the statement in LINENUMBER, and the last line in LINENO.

If an error occurs in the lexer, LEXCOUNT is set to 0.

Each **element** of **the** array LEXEME is a record with three pieces of information:

1. LEXEME.T: The type of the lexeme.
2. LEXEME.F: The index in LEXSTRING of the first character of this lexeme.
3. LEXEME.L: The index of **the** last character of this lexeme.

For example, if the identifier COMMON occurs in columns 7 to 12 and it is the first lexeme of the statement (**the** label is not counted as a lexeme), then the entries in LEXEME will be

```
LEXEME[1].T = IDENTIFIER
LEXEME[1].F = 7
LEXEME[1].L = 12
```

4.2 Lexemc *types*

A **lexeme** is defined to be one of **the** following items:

name	description
PLUS	+ sign
MINUS	- sign
STAR	*
SLASH	/
EXPONENT	**
LPAREN	(
RPAREN	)
EQUALS	.
COMMA	,
LE,LT,GE,GT	.LE.,.LT.,.GE.,.GT.
EQ,NE	.EQ.,.NE.
ANOOP , OROP	.ANO., .OR.
NOTOP	.NOT.
REALCON	a Fortran real constant (not including preceding sign)
OPCON	double precision const (not including preceding sign)
INTEGERCON	an integer constant (not including sign)
STRINGCON	quoted or Hollerith constant
TRUECON	.true.
FALSECON	.false.
IDENTIFIER	a sequence of characters, the first of which must be a letter and the rest may be letters or numbers

EXPLMARK	!
VUOTMARK	"
NUMSIGN	#
DOT	.
DOLSIGN	$
PERCENT	%
AMPE RSANO	6
COLON	.
SEMICOLON	;
LESSSIGH	<
BIGGERSIGN	>
QUESMARK	?
ATSYM	@
LSQBRACKET	[
RSQBRACKET	]
BACKSLASH	\
CARET	↑
EOS	end of statement
NON	none of the above

4.3 Reading in a statement

When LEXER is called, LEXSTRING is cleared by putting In blanks. It then invokes the procedure GETSTATEMENT to load the characters of the next statement into LEXSTRING. It assumes that the first six characters of the next line are already in the array COL1TO6. If the first letter is "C", then the line is a comment line. COL1TO6 is printed in the listing file and the comment itself is read into the listing file (procedure SKIPLINE). The variable LINENO is used to keep track of the number of lines that are read in.

As soon as a non-comment line is read in (this may be a blank line), the global variable LINENUMBER, which always contains the line number of the first line of the current statement, is set to LINENO. If the end of file has been reached, this is indicated by setting LEXSTRLENGTH to 0. COL1TO6 is copied to both the listing file and LEXSTRING. The rest of the statement is read in, putting each character in both the listing file and LEXSTRING, until the end of the statement is encountered. If comment lines occur, they are skipped over as previously. Continuation lines are recognized and appended. To determine this, GETSTATEMENT must always look ahead to the next 6 characters of the next line. Thus at the end of GETSTATEMENT, the first 6 characters of the next non-comment line will be in COL1TO6. Each line is padded with blanks so that it always is 72 plus a multiple of 66 characters in length. After a statement is read in, LEXSTRLENGTH will contain the number of characters in LEXSTRING. At this point, LEXSTRING is aiso written to the U-Code file by procedure PRINT-LEXSTRING.

After LEXER calls GETSTATEMENT, it checks to see if the statement returned consists only of blanks. If it does, it calls GETSTATEMENT again. In this way, blank lines are allowed. Next, it checks to see if the first 6 characters of LEXSTRING contain a label. If it does, this label is converted to an integer and stored in the global variable LABNO.

4.4 Scanning the statement

Next, the array LEXEME is filled with lexemes that are recognized through a case statement based on the first characters of the lexemes inside a WHILE loop that traverses the LEXSTRING array. The procedure NEXTCHAR is generally used to get the next character. But since it skips blanks, it is not used in processing identifiers, numbers and keywords.

If the first character of the lexeme is a regular Fortran character other than a letter, digit, single quote or dot, then the lexeme type is set to that character. (In the case of an asterisk, the next character must be checked to see if it is a double asterisk.)

If it is a digit, then the procedure SKIPDIGITSTRING finds the last digit. If the digit string is followed by an H, then the lexeme is a Hollerith string. If it is followed by a dot, then it may be either a real or an integer followed by a dot-word (as in "33. EQ. X"). The procedure FINDWORD is called to get the character string if it is a dot-word. (If this is the case, it results in two lexemes being processed in a single pass: the integer and the dot-word). If the dot is not followed by a letter, DIGITSTRING is called again to find the last digit of the fraction of the real number, and then FINDEXPONENT to get the exponent If the first digit string is followed by neither a dot nor an H, then the lexeme is an integer.

If the first character is a dot, then the lexeme is either a dot-word or a real (again, FINDWORD and FINDEXPONENT are used).

If the first character is a single quote, then the lexeme is a string. When an embedded quote occurs in a string constant, one of the two quotes is deleted and the string content to the left is shifted right by one position. This is because after LEXER, the compiler will use the information in the array LEXEME to determine the extent of the string constant.

If the first character is a letter, then the lexeme is an identifier, and characters are skipped until the next non-alphanumeric letter is read in. The identifier FORMAT is recognized as a reserved word and it is processed as a special case. The FORMAT specification, including both surrounding parentheses, is processed as a string constant. Consequently, the name FORMAT cannot be used as the name of a variable.

Blanks are skipped everywhere, except in identifiers, numbers and key words.

The syntax for lexemes is described below using Wirth's variant of BNF:

```
lexeme = special-symbol | dot-word | number | Hollerith |
         identifier.

special-symbol = "+" | "-" | "*" | "/" | "(" | ")" | "=" | "**" |
                 "," | "!" | """ | "#" | "$" | "%" | "&" | ":" |
                 ";" | "<" | ">" | "?" | "@" | "[" | "]" | "/" |
                 "↑"

dot-word = ".LE."|".LT."|".GE."|".GT."|".NE."|".EQ."|
           ".AND."|".OR."|"'NOT."|".FALSE."|".TRUE.".

number = mantissa [exponent].

mantissa = digit-string "." [digit-string] |"." digit-string.

digit-string = digit {digit}.

exponent = ("D"|"E")["+"|"-"] digit-string.

Hollerith = digit-string "H" {character} | "'" {character} "'".

identifier = letter {letter|digit}.
```

6. Statement Classifier

Once a statement has been read in by LEXER, it is determined to be one of the following types by procedure CLASSIFY:

```
STATEMENT-CLASS = (XNONE,XARITH,XASSIGN,XLOGICALIF,XARITHIF,XGOTO,
           XCALL,XRETURN,XEND,XPRINT,XBLOCKDATA,XFORMAT,XSET,XOPEN,
           XCONTINUE,XSTOP,XPAUSE,XDO,XREAD,XWRITE,XREWIND,
           XBACKSPACE,XENDFILE,XEXTERNALFUNC,XSUBROUTINE,XENTRY,

           XDIMENSION,XCOMMON,XEQUIVALENCE,XIMPLICIT,
           XEXTERNAL,XLOGICAL,XINTEGER,XCOMPLEX,XREAL,XDOUBLE,
           XDATA,XINTERNALFUNC);
```

CLASSIFY first checks to see if the statement is an assignment statement or statement function declaration, since keywords such as DO and GOTO are legal variable names. If the statement is of the form:

identifier = anything

or

identifier (anything) = anything

then it is one of the two. In the second case, if the symbol is a dimensioned array (all DIMENSION statements must occur before all statement function declarations), then the statement is an assignment statement; otherwise it is a statement function declaration.

If the statement is not an assignment statement or a statement function, then the first lexeme of the statement is compared with all keywords of the same length. Normally, the statement type is determined right there. The oniy exceptions are:

For INTEGER, REAL, COMPLEX, or LOGICAL, the next lexeme is checked to see if it is the identifier FUNCTION, and the lexeme further down an identifier, since FUNCTION can be used as the name of a variable.

For DOUBLE, the next lexeme is checked to make sure it is the identifier PRECISION.

For BLOCK, the next lexeme is checked to make sure it is DATA

For IF, CLASSIFY determines whether the statement is an arithmetic or logical IF. An IF statement is an arithmetic IF if it is of the form

IF **(anything) number anything**

Otherwise, it is a logical IF. (While scanning between the parentheses, both in this case and while checking to see if the statement is an assignment statement, it is necessary to keep track of the number of left and right parentheses in order to allow for nested parentheses.)

If the current statement already has error discovered in LEXER, it wilt be classified as XNONE. When CLASSIFY finds any erroneous construct, it will also classify the current statement as XNONE. CLASSIFY outputs no error message.

6. Main block

The processing of an input user program is controlled by the main procedure and procedure BLOCK The control structures of these two procedures are as follows:

6.1 Main procedure

1. Call INITCOMPILER to initialize everything.

2. Call BLOCK to process the main program unit.

3. While there are more subprograms do
   (a) call FUNC STMT, SUBR_STMT or BLKDATASTMT to process the heading of the next program unit;
   (b) call BLOCK to process body of program unit.

4. Call VARINITIALIZATION to generate the code to initialize the variables that should be initialized and to load FORMAT specifications into memory.

5. Generate the bodies of the level 1 to 3 dummy U-Code procedures.

6.2 *Procedure BLOCK*

1. (a) Call LEXER to get the first statement of the current program unit;
   (b) Call CLASSIFY to determine the statement type.

2. If first statement is the IMPLICIT statement,
   (a) call IMPLDECL to process it;
   (b) call LEXER to get the next statement;
   (c) call CLASSIFY to determine the statement type.

3. While there are more declaration statements, FORMAT or SET statements do
   (a) call the appropriate routine to process it;
   (b) call LEXER to get the next statement;
   (c) call CLASSIFY to determine the statement type.

4. Call STORAGE_ALLOCATION to allocate storage for the variables that have been declared.

5. Call FILL_ADDRESS_INITIALIST to copy these addresses into the list of variabies to be initialized.

6. While there are statement function declarations, FORMAT or SET statements do
   (a) call STMT_FUNCTION, FORMAT_STMT or SET_STMT;
   (b) call LEXER to get the next statement;
   (c) call CLASSIFY to determine the statement type.

7. Generate code for the head of the U-Code procedure for the current program unit.

8. Initialize the list of temporary locations to **NIL.**

9. While statement is an executable statement, FORMAT or SET statement do:
   (b) if there is a Fortran label, enter it in the label table if it is not there already and generate code for a U-Code label by calling ENTERLABEL;
   (c) call the routine to process the statement;
   (d) if we are not about to process a statement within a logical IF statement then do
      (1) if we have been processing an IF statement, then generate the U-Code label to be jumped to if the condition is false;
      (2) if there is a Fortran label and it is the end for a do-loop, then generate the appropriate code;
      (3) call LEXER to get the next statement;
      (4) call CLASSIFY to determine the statement type;

10. (a) Process the END statement;
   (b) call LEXER to get the next statement;
   (c) call CLASSIFY to determine the statement type.

11. Check if any do-loop is still open.

12. Check the label and symbol tables and issue warnings if any label or variable have been used only on the left-hand-side or only on the right-hand-side.

13. Generate code for the end of the U-Code procedure for the current program unit.

7. Symbol Tables

7.1 The structure of the tables

There are five symbol tables in UFORT:

1. The main *symbol* table keeps track of variables, subprogram and entry names, intrinsic and standard function names and FORMAT labels used within a single program unit (main program or subprogram).

2. The label table keeps track of Fortran labels within a single program unit.

3. The *common name* table keeps track of common areas.

4. The *external name* table keeps track of subprogram and entry names throughout ail the program units.

5. The *standard function* table contains the names of all standard functions.

Each of these tables is made up of records which form a binary tree. The symbols are ordered lexicographically in the tree. The heads of the tables are pointed to by pointers stored in the global variables SYMHEAD, LABELHEAD, COMHEAD, EXTHEAD, and HEADSTDTABLE.

The main symbol table and the label table are cleared at the beginning of each new program unit. The other three are cleared only once, at the beginning of compilation. The storage used by the cleared entries is automatically reclaimed through the garbage collection facility in &he Pascal in which UFORT is written.

7.2 The associated routines

The standard function table is set up at compiler initialization time and has a routine, IN_STNDFUNCTABLE, that searches it. The other four each has a main routine that searches the table for a given entry and inserts it in if it is not already there, and then adds any information to the symboi table that is not contradictory to the information it already has about these symbols. This structure is convenient in a one-pass Fortran compiler, because the information for a symbol is typically scattered all over the program.

The four main routines, called FSYMBOL, FLABELNO, FCOMNAME, and FEXTNAME, are very similar in structure, and have similar subsidiary routines which they call, For example, the routines CLEARSYMBOL, CLEARLABELNO, CLEARCOMNAME, and CLEAREXTNAME all initialize new records for insertion into the respective table. The following description of how procedure FSYMBOL works, therefore, is applicable to the other three routines.

When FSYMBOL is called, it calls procedure BUILDSYMBOL with a name and a pointer to the head of the table as parameters. BUILDSYMBOL, which uses procedure SYMLOOK, searches for an entry in the table with that name. If it does not find the symbol, it will create a new record and procedure CLEARSYMBOL will be called to set the fields of the record to their default values. FSYMBOL then inserts all the information about this symbol that was passed to it as parameters, checking for contradictions with the information it already has. It is assumed that contradiction does not exist among the call parameters in a single call.

The four symbol table routines FSYMBOL, FLABELNO, FCOMNAME and FEXTNAME can be used for 3 different purposes: (a) to retrieve the pointer to the symbol table entry, (b) to assert information about the symbol as given in the parameters in the call, and (c) to test the properties of the symbol against the values given in the parameters in the call. Each of the routines depart from (c) somewhat, and the details are given in their sections following.

7.3 The main symbol table

The main symbol table stores information about the characteristics of the identifiers used in a block, the most important of which are their addresses. It also stores the FORMAT labeis. A space in memory for saving the address of the FORMAT string is allocated for each FORMAT label (see Section 25).

It uses records of type SYMBOL:

```
DIM = RECORD CASE INTEGER OF (* array dimension *)
          0:(CONSDIM:INTEGER); (* constant *)
          1:(VARDIM:↑SYMBOL); (* variable *)
         END ;

FUNCTYPE = (NOTEXTERNAL ,EXTERNAL,EXTSUBR,EXTFUNC,STMTFUNC,
           INTRINSTDEXT,PARAMPROC);

SYMBOL = PACKED RECORD
         LSON,RSON:↑SYMBOL;       (* POINTERS TO SONS *)
         NAME : THENAME;          ( * SYMBOL NAME, 6 CHARACTERS LONG * )
         STYPE :POINTDEFTYPE;     (* THE TYPE OF THE VARIABLE; IT SHOULD
                                     BE SET TO NONE IF SUBROUTINE NAME *)
         WHEREDEFINED:INTEGER; (* PROGRAM LINE NUMBER IN WHICH
                                     VARIABLE APPEARS THE FIRST TIME *)
         LEVEL,                   (* ADDRESSING LEVEL FOR THE VARIABLE *)
         ADDRESS:INTEGER;         (* -1 IF NOT YET ESTABLISHED. *)
         MTYPE : CHAR ;           ( * CHARACTER FOR THE MEMORY TYPE * )
         USED-LHS,                (* TRUE IF VARIABLE WAS GIVEN A VALUE,
                                     NOT USED FOR EXTERNL,EXTSUBR,
                                     INTRINSTDEXT, PARAMPROC, EXTFUNC,
                                     EXCEPT WHEN A FUNCTION VARIABLE * )
         USED_RHS,                (* TRUE IF VARIABLE'S VALUE WAS USED,
                                     NOT USED FOR INTRINSTDEXT,
                                     FORMATLABEL * )
         S-DUMMY,                 (* TRUE IF DUMMY ARGUMENT * )
         S-EXPLICIT: BOOLEAN;     ( * TRUE IF TYPE EXPLICITLY DECLARED * )
   CASE S_FUNCSUBR: FUNCTYPE OF (* NOTEXTERNAL IF NOT EXPLICITLY ASSERTED *)
       INTRINSTDEXT:(PTRSTD:↑STDFUNCTABLE); (* POINTER TO STANDARD FUNCTION
                                              TABLE IF STARDARD FUNCTION NAME *)
       STMTFUNC:    (SEGMENNUM, (* SEGMENT NUMBER OF ITS U-CODE PROC BLOCK * )
                    NUMOFARG: INTEGER);
       NOTEXTERNAL: (S1_EQUIVALENCE,       (* TRUE IF EQUIVALENCED * )
                     S2_EQUIVALENCE,       (* USED TO INDICATE IF AN EQUIV.
                                              VARIABLE HAS BEEN PROCESSED IN
                                              STORAGE ALLOCATION TO CHECK
                                              EOUIVALENCING TWICE *)
                     S-COMMON,             ( * TRUE IF COMMON VARIABLE * )
                     INITIALIZED: BOOLEAN; (* TRUE IF VARIABLE INITIALIZED.
                                              FALSE OTHERWISE *)
                     (* FOLLOWING FIELDS DO NOT HAVE CORRESPONDING PARAMETER IN
                        PROCEDURE FSYMBOL *)
                     NUM_ELEMENTS: INTEGER; (* ONE IF SCALAR; ELSE NUMBER OF
                                              ELEMENTS IN ARRAY, ZERO IF
                                              ADJUSTABLE DIMENSION *)
                     PTRCOM:↑COMNAME;      (* POINTER TO THE COMNAME TABLE,
                                              USED ONLY IF COMMON SYMBOL *)
                     ARRY:POINTARRY_INFO);  (* POINTER TO TABLE Of DIMENSIONS,
                                              NULL IF NOT AM ARRAY * )
          END;
```

```
ARRY_INFO = PACKED RECORO
          DIMENSION: INTEGER;        (* THIS MUST NOT BE 0 *)
                                     (* FOLLOWING 2 ARRAYS USED ONLY UP TO
                                        'OIMENSION' *)
          DIMEN:ARRAY[1..MAXDIM] OF DIM;(* EITHER THE CONSTANT
                                  OIMENSION OR THE POINTER TO THE SYMBOL
                                  TABLE ENTRY IF VARIABLE OIMENSION *)
          S_CON:ARRAY[1..MAXDIM] Of BOOLEAN; (* TRUE IF THE ITH
                                        OIMENSION IS CONSTANT *)
          NUM_CONST_DIMS,            (* THE FIRST 'N_C_D' OF S-CON ARE TRUE *)
          FIRST-OFFSET: INTEGER; (* CORRECTION FOR 'COMPUTE OFFSET' *)
        END;
```

Its main procedure, FSYMBOL, has parameters that correspond to the record fields whose contents are **checked** inside this procedure.

```
PROCEDURE FSYMBOL(VAR SPTR:POINTSYMBOL; (* RETURNS ALWAYS A POINTER TO THE
                                           ENTRY IN THE SYMBOL TABLE *)
                  SYMNAME: THENAME;
                  SYMTYPE:DATATYPE;        (* NONE IF NO INFO IS SENT *)
                  SYMWHEREDEFINED:INTEGER; (* THIS WILL CONTAIN THE PROGRAM
                                           LINE NUMBER BEING PROCESSED ●  )
                  SYMFUNCSUBR:FUNCTYPE;    (* NOTEXTERNAL IF NO INFO, THE
                                           PROPER FUNCTYPE OTHERWISE *)
                  SYMCOMMON ,
                  SYMDUMMY,
                  SYMEQUIVALENCE,
                  SYMLHS,
                  SYMRHS,
                  SYMINITIALIZED:BOOLEAN); (* FALSE IF NO INFO OR FALSE *)
```

Most of **the** entries in this symbol table assume an implicit value if no information is asserted. When it is necessary to check that an entry is having a certain **value, it** is possible to accomplish the check by asserting the entry to that value using the corresponding parameter in the call to FSYMBOL. Note **that** in this case, if the entry is having **the** implicit **value,** it **will be** changed to the **asserted value,** which is undesirable in **some** cases. When the check is for the entry to **have** the **implicit** value, this does not work, since **the** implicit value in the call parameter specifies **no** action. **Thus,** it is necessary sometimes to retrieve the **pointer and** then make the comparison explicitly.

If STORAGE_ALLOCATION has already **been called,** i.e. **when** processing the executable part of a program unit, FSYMBOL allocates space for **new** variables not previously declared using procedure SIMPLE-STORAGE If no allocation is desired (e.g. when testing that **a** statement function name has not previously been declared as a variable), BUILDSYMBOL should be used to retrieve the pointer rather than FSYMBOL.

Field S-EXPLICIT is set to true whenever STYPE has been asserted in a call. FSYMBOL will **automatically** infer **a** symbol to be EXTFUNC if it is both typed **and declared** EXTERNAL

See Section 15.2 regarding the S_FUNCSUBR field.

7.4 The label number table

Both statement labels and FORMAT labels are entered into this table. For each statement label, it also stores the U-Code label associated with it. This association is fixed the first time the Fortran **label** occurs in **the** program unit, **when the** new table entry is created. The position of the label in the statement, i.e. whether it is on the left-hand side ("100 X=1") or the right-hand side ("GOTO 100"), is kept in the table.

The label number table is made up of records of type LABELNO:

```
LABELTYPE = (LNONE,ISFORMAT,ISSTMT);

LABELNO = PACKED RECORD
            NAME,                    (* FORTRAH LABEL *)
            PLABEL:INTEGER;          (* PCOOE LABEL NUMBER ASSOCIATED *)
            LSON,RSON:↑LABELNO;
          'IS-ON-RHS,
            IS_ON_LHS:BOOLEAN;       (* TRUE IF THIS LABEL NUMBER HAS OCCURRED
                                        ON RIGHT/LEFT HAND SIDE OF STATEMENT')
            LTYPE : LABELTYPE;       (* TELLS WHETHER A FORMAT OR STATEMENT
                                        LABEL .  NONE WHEN FIRST CREATED *)
          END;
```

and is accessed by the routine FLABELNO:

```
PROCEDURE FLABELNO (VAR LPOINTER:POINTLABELNO;
                    NUMBER:INTEGER;      (* FORTRAN LABEL *)
                    L IS-ON-RHS,
                    L IS_ON_LHS: BOOLEAN;  (* FALSE IF NO INFO OR FALSE *)
                    LABTYPE: LABELTYPE); (* TYPE OF LABEL, MUST BE ASSERTED *)
```

Places where FLABELNO is called are procedures ENTERLABEL called by BLOCK, COMPLUJP and COMPLFJP used in the GOTO and arithmetic IF statement processors, the DO statement processor and the READ/WRITE statement processor.

7.5 *The common table*

The common name table (COMNAME) simply stores the names of the common areas thus far defined and some information about them. It is made up of records of type COMNAME:

```
COMNAME = PACKEO RECORO
            LEVEL,                   (* PSEUDO LEVEL NUMBER FOR THIS COMMON
                                        AREA *)
            LENGTH,STADOR:INTEGER;   (* LENGTH OF THE COMMON BLOCK IN OUARTER
                                        WORDS AND STARTING ADDRESS *)
            PTRCOMLIST:↑COMLIST;     (* POINTER TO THE LINKED LIST Of COMMON
                                        ELEMENTS IN THIS AREA *)
            LSON,RSON: ↑COMNAME;
            NAME : THE NAME ;        ( * NAME OF THE COMMON AREA * )
          END;
```

and accessed by the routine FCOMNAME during storage allocation:

```
PROCEDURE FCOMNAME (VAR CPOINTER:POINTCOMNAME; CONAME:THENAME);
```

LEVEL is initialized inside CLEARCOMNAME, immediately after the entry is created. PTRCOMLIST, which points to a linked list of variables, is built when processing the declarations of the corresponding COMMON area. At the beginning of each program unit, the field PTRCOMLIST of ail entries is set to NIL.

When an entry is first created for a common area name, LENGTH is set to the value given by global variable COMMONSIZ. This variable has a default value 0, and is set by the option CSIZ. At the end of processing a COMMON statement, this variable is reset to 0. When space IS allocated the first time for a common area, if the actual allocated area is greater than that specified in LENGTH, this field is changed to the larger value. Otherwise, the amount of space allocated is equal to the value of LENGTH. Thereafter, its value is fixed.

STADDR, initially set to -1, indicates whether a memory block has been allocated to the common area in a previous program unit. If yes, it gives the start address of this block.

FCOMNAME is called only in the common statement processing procedure. It only returns the pointer to the common table entry. During storage allocation, the entries are accessed by traversing the tree.

7.6 The external name table

The external name table keeps track of the existence and calls of the various subprograms. An entry in the external name table implies the existence of a subprogram with that name. A symbol can be in the EXTNAME table and in the SYMBOL table at the same time, when the external subprogram name is referenced in the program unit, or there is an internal variable or statement function name which happens to have the same name as another subprogram. When processing a subprogram, the subprogram name is also in both tables, and in the case of function subprograms, the name is used internally as a function variabie.

An identifier declared EXTERNAL is not necessarily entered in the external name table. (See Section 8.7.)

A symbol is inserted in the external table when it is called, defined or passed as a subprogram name parameter. This occurs in (a) procedure USERFUNC, which processes calls, (b) the FUNCTION statement processor, (c) the SUBROUTINE statement processor, (d) the ENTRY statement processor and (e) procedure PROCESS_ARGUMENTS.

The table is made up of records *of* type EXTNAME:

```
EXTNAME = PACKED RECORD
                    LSON,RSON: ↑EXTNAME;
                    NUHBER : INTEGER;      (* SEGMENT NUMBER ASSOCIATED TO THIS
                                              SEGMENT NAME ENTRY *)
                    XFUNCSUBR: FUNCTWE; (* MUST BE ONE OF EXTFUNC. EXTSUBR,
                                              NOTEXTERNAL • )
                    TYPEEXPL ICIT,         (* TRUE IF EXPLICIT TYPE IN SUBPROGRAM
                                              HEADING ● )
                    IS-DEFINED,            (* A SUBPROGRAM BLOCK EXISTS FOR IT *)
                    IS-CALLED,             ( • INVOKED AT LEAST ONCE * )
                    IS_PASSED: BOOLEAN;    (* HAS BEEN PASSE0 AS PARAHETER *)
                    STYPE:POINTDEFTYPE; (* THE TYPE Of THE FUNCTION; IF
                                              SUBROUTINE, THIS FIELD NOT USED *)
                    NAME:THENAME;
                    NUMOFARG: INTEGER;     (* NUMBER OF ARGUMENTS; -1 IF NO INFO *)
           END;
```

and accessed by the routine FEXTNAME:

```
PROCEDURE FEXTNAME (VAR EPOINTER:POINTEXTNAME;
                        EXNAME:THENAME;
                        EXTYPEEXPLICIT: BOOLEAN;(* TRUE IF EXPLICIT TYPE IN
                                                   SUBPROGRAM HEADING *)
                        EXTYPE : OATATWE ;         (* NONE IF NO INFO *)
```

```
                    EXFUNCSUBR: FUNCTYPE;      (* NOTEXTERNAL IF NO INFO *)
                    EXDEFINED,
                    EXCALLED: BOOLEAN);        (* FALSE IF NO INFO *)
```

NUMBER is filled automatically inside CLEAREXTNAME immediately after the external name table is created, in such a way that each external program unit is associated with a different segment number.

FEXTNAME is designed both for asserting and checking. This is because it is not sure when the mode is assertion and when it is checking, since the position of a subprogram bears no relationship to where its calls originate. FEXTNAME checks the STYPE and XFUNCSUBR fields if the external symbol is either previously called or defined. Otherwise, it goes ahead to assert STYPE and XFUNCSUBR to the values given in the parameters.

When FEXTNAME is called from (a), parameter EXTYPE is to be the STYPE Value of the symbol's entry in the symbol table, even if its type is implicit, since the type in the external table is fixed after the first call.

When FEXTNAME is called from (b) or (c), parameter EXTYPEEXPLICIT indicates whether typing is explicit in the FUNCTION statement. This is needed because FEXTNAME is called once again before processing the first statement, or after processing the IMPLICIT statement if present as the first statement in the subprogram. This call is from procedure BLOCK. The pointer is retrieved. If the TYPEEXPLICIT field is false, then if the subprogram has been called, check is made against the now known implicit type. Otherwise, the implicit type is assigned.

7.7 The standard function table

The standard function table is initialized by the procedure FILL_STDNNCTABLE It is made up of the following type of record:

```
STDFUNCTABLE = RECORD
                   NAME : THENAME;
                   NUMBER: INTEGER;     (* EACH PROCEDURE HAS A DIFFERENT
                                           NUMBER,USED WHEN THE FUNCTION
                                           IS CALLED *)
                  LSON,RSON:↑STDFUNCTABLE;
                 END;
```

It is searched by the function IN_STDFUNCTABLE:

```
FUNCTION IN_STDFUNCTABLE(NAME:THENAME;VAR STDPTR:POINTSTDFUNCTABLE):BOOLEAN;
```

8. Processing of Declarations

When a variable occurs in a declaration, an entry for that variable is made in the symbol table by calling procedure FSYMBOL, and the information given in the declaration is filled in. An error message is issued if that symbol already has some contradictory information. The address of the variable is not determined at that time, because when a declaration is scanned, not all the information about the variables is known. The assignment of an address to the variable declared occurs in procedure STORAGE_ALLOCATION (see Section 11).

8.1 Representation of types

The numerous data types which the compiler recognizes are represented in records defined as follows:

```
DATACLASS = (INTEGERCLASS, REALCLASS, COMPLEXCLASS, LOGICALCLASS,
             STRINGCLASS, OTHERCLASS);

POINTDEFTYPE = ↑DEFTYPE;

DEFTYPE = RECORD
            SIZE: INTEGER;
            GENTYPE: CHAR;
            CASE CLASS: DATACLASS OF
            COMPLEXCLASS : (COMPPART : ↑DEFTYPE);
          END;
```

The different data types are represented by pointers to their own individually-defined records. The pointer variables are named after the type names, and they are globally defined and initialized in procedure INITCOMPILER (see Section 6). This structure allows easy access to the size, U-Code type and class of each data type. In the case of the types for complex numbers, an additional pointer field in this record points to the type of the real and imaginary component parts.

The data types used in the compiler are:

LOGICAL1, LOGICAL2, LOGICAL4, LOGICAL8 - for booleans;

INT 1, INT2, INT4, INT8 - for integers;

RE1, RE2, RE4, RE8 - for real numbers;

COMP4, COMP8 - for complex numbers;

STRING - for string constants;

FORMATLABEL - for labels of FORMAT statements;

NONE - for the data type of subroutines;

POINTER - for addresses (the U-Code type A);

SINGCHAR - for a single character (U-Code type C);

PROC - for procedures (the U-Code type P);

SINGSET, DOUBSET - for the U-Code set types.

8.2 Type-specific declarations

Procedure TYPEDECL scans and processes this kind of declaration. Variables are inserted in the symbol table with the information specified by the declaration.

First, it obtains the type for the variable, based on the type of the declaration. It then scans forward and obtains its size modified by "*" if one is specified. The variable is inserted in the symbol table and a pointer to the symbol table entry is passed to procedure ISARRAY. This procedure is responsible for obtaining the dimension information for creating the record that stores this information and putting its pointer in the symbol table entry of the variable.

If the variable is initialized, procedure VARINIT is responsible for the steps involved. This procedure builds a list of the variables to be initialized. (See Section 9.)

VARINIT is entered with LXC (the global pointer to the lexeme array) pointing to the lexeme with the first initialization value. The initialization list is extended at the end by calling EXTEND_LIST a number of times according to the number of elements in the variable declared. Procedure FILL_VALUES is then called which traverses the list of the initialization values in the statement and enters them into the fields of the nodes just created. In this process, it calls procedure INSERT-VALUE.

Procedures EXTEND-LIST, FILL-VALUES and INSERT_VALUE are also used in processing the DATA statement. See Section 9.2.

8.3 Dimension declaration

Procedure DIMENDECL scans and processes the Fortran DIMENSION statement. The symbol table entries for the variables are updated with the dimensron information, It uses procedure ISARRAY to obtain the dimension information as in type-specific declarations.

8.4 Implicit declaration

Procedure IMPLIDECL scans an IMPLICIT statement. Array IMPLIARRAY is filled with the specified implied types. IMPLIDECL can be entered only when processing the first statement in a program unit.

This procedure gets the implied types and size modifications, and inserts them in IMPLIARRAY for the list of letters specified, using procedure LETTERLIST. If an IMPLICIT statement occurs in a subprogram, the dummy arguments are affected plus the function name if it is a function subprogram. Therefore, once all the declarations are scanned, the symbol table entry is traversed in order to change the standard Fortran implied types for the dummy arguments and function names, using procedure CHANGEDEFAULTS. These are the only valid symbols in the symbol table at that time because the IMPLICIT statement must be the first statement in a program unit.

8.5 *Common* declaration

Procedure COMDECL scans and processes a common declaration, The common name table is built inside this procedure and linked-lists of the common variables in each common area are constructed. This list is formed with COMLIST records that have the following format:

```
COMLIST = RECORD STPTR:↑SYMBOL;    (* POINTER TO SYMBOL TABLE ENTRY OF
                                      COMMON ELEMENT *)
                 NEXT: ↑COMLIST;
              END;
```

The root of the list of common variables for each common area is stored in the field PTRCOMLIST of its entry in the common name table.

For each common area, COMDECL first gets its name and inserts it in the common name table. If it is already in the table, it obtains the last entry in the common variable list for that area. Using this pointer, the declared variables in this area are inserted in the order they are declared. These variables are also entered in the main symbol table, if necessary, along with the information that they are in a common area fields (S-COMMON is set to TRUE, and PTRCOM is set to point to the correct entry in the common table).

Any dimension information of a variable in a common declaration is treated as dimension declaration, and this information is obtained with procedure ISARRAY.

Information about the length and starting address of the common areas is not inserted here but in procedure STORAGE-ALLOCATION, where the addresses for the common variables are assigned. The reason for this is that a variable may be dimensioned in a later statement, so there is no way to be sure how much space it will take until all the declarations have been processed.

The blank common area is called "M M M " internally in the compiler. The spaces between the M's make it impossible for any user to use this name as a name for one of its common areas.

8.6 Equivalence declaration

Procedure EQUIVALDECL scans and processes EQUIVALENCE declarations. This procedure builds the list of equivalence groups and it also builds the circular lists of equivalenced variables that form the equivalence groups.

The list of equivalence groups is formed with EQGROUP records and the lists of equivalenced variables are formed with EQLIST records.

```
EQGROUP = PACKED RECORD
              LOW,HIGH:INTEGER; (* STORE THE LOWER AND HIGHER BOUNDS
                                   OF THE EQUIVALENCE GROUP * )
              LEADER:↑EQLIST;   ( • POINTS TO FIRST ELEMENT IN LIST OF
                                   EQUIVALENCE VARIABLES THAT FORM 6ROUP *)
              NEXT : ↑EQGROUP;  ( * POINTS TO NEXT GROUP * )
              ALLOCATED,        (* TRUE IF THE 6ROUP HAS ALREADY BEEN
                                   ALLOCATED IN MEMORY ●    )
              HAS_INIT,         (* HAS ONE VARIABLE INITIALIZED *)
              HAS_COMMON:BOOLEAN; (* TRUE WHEN THIS GROUP HAS
                                     A COMMON ELEMENT. *)
          END;

EQLIST = RECORD STPTR: ↑SYMBOL; .
```

```
                    ( • POINT TO SYMBOL TABLE ENTRY OF EQUIVALENCEO VAR. * )
                DIMENSION:ARRAY[1..MAXDIM] OF INTEGER;
                    (* USED TO STORE THE COORDINATES OF ARRAY ELEMENT
                       EQUIVALENCED * )
                OFFSET:INTEGER;
                    (* OFFSET OF THE ELEMENT WITH RESPECT TO THE LEADER OF
                       THE LIST * )
                NEXT: ↑EQLIST;
                    (* NEXT IN THE LIST * )
          END;
    (* THIS LIST IS USED TO STORE THE VARIABLES THAT ARE EQUIVALENCED
       IN ONE' EQUIVALENCE GROUP *)
```

For each equivalence group, procedure EQUIVALDECL calls procedure EQUIVARLIST. This procedure gets the names of the variables that form the group, inserts them in the symbol table, if required, setting field S1_EQUIVALENCE to TRUE, and inserts them in the circular list that form the equivalence group. If the variable equivalenced is an element of an array, its coordinates are also obtained. All this is done inside procedure EQUIVARLIST.

With the equivalence groups declared, a list is formed using the global variable EQUIVHEAD that points to the head of the list and TAILEQGROUP that points to the most recently declared equivalence group at the tail.

Since the coordinates for array elements are remembered instead of being processed immediately, dimension declaration of a variable can occur after its EQUIVALENCE statement.

8.7 External Declaration

Procedure EXTDECL scans and processes an external declaration. The information that a variable is external is entered in the symbol table only, since the effect of the external declaration is restricted to inside its program unit. The external table is updated later in the call to the external symbol, when the existence of a program unit of that name is implied. Information is not entered in the external table if the variable externalled is a dummy argument.

9. Initialization of Variables

In most Fortran compilers, initializations are handled by setting up the binary load file so that the locations which are specified by the variables to be initialized are loaded with the initial values at the time the program is loaded. It is not possible to do this in U-Code, since storage IS allocated on the stack only when the corresponding procedure is entered; instead, a series of explicit loads and stores must be executed at the beginning of the program.

The initialization of variables consists of three stages. First, a list of the variables to be initialized is formed during the processing of type-specific declarations (Section 8.2) and DATA statements. Next, the addresses of the variables to be initialized are saved in the LEVEL and ADDRESS fields of the record entries in the initialization list when procedure FILL_ADDRESS_INITIALIST is called after storage allocation for the current program unit has occurred. Finally, code are generated for the initializations at the end of compilation by calling procedure VARINITIALIZATION.

9.1 The initialization list

This linked list containing the variable addresses to be initialized and their initialization values is formed using the INITIALIST record with the following structure:

```
INITIALIST = PACKED RECORD
                    SYMTABPTR : ↑SYMBOL;   (* POINTER TO SYMBOL TABLE ENTRY
                                              OF VARIABLE TO BE INITIALIZED *)
                    LOCSIZE:INTEGER;       (* SIZE OF INITIALIZED LOCATION;
                                              FOR COMPLEX, SIZE OF EACH HALF * )
                    NEXT:↑INITIALIST;      (* NEXT NODE *)
                    LEVEL,                 (* LEVEL OF THE VARIABLE *)
                    ADDRESS:INTEGER;       (* LOCATION TO BE INITIALIZED.
                                              EVEN IF ARRAY ELEMENT *)
                    AMOUNT:DIGIT_STRING;   (* STRING WITH THE VALUE
                                              TO BE INITIALIZED *)
                    CONTINUING:BOOLEAN;    (* TRUE IF THIS IS A CONTINUUM OF
                                              THE PREVIOUS NODE, USED IN
                                              INITIALIZATION WITH STRINGS *)
                CASE AMOUNTYPE : LEXTYPE OF (* LEXTYPE OF THE STRING VALUE *)
                STRINGCON:
                    (STRLEN : INTEGER) ;   (* IF INTIALIZATION WITH STRING,
                                              LENGTH OF THE STRING CONSTANT *)
                INTEGERCON,REALCON,DPCON:
                    (NEGATIVE:BOOLEAN);     (* TRUE IF CONSTANT IS -VE * )
                END;
```

The same initialization list is used for all the program units in a program, lengthening as more initializations are specified. The addresses have to be saved in this list because the symbol tabies of all previous program units are no longer available when the initialization code is being emitted in procedure VARINITIALIZATION.

One entry is created for a simple variable. Complex vartables are inserted in the list of initraiized variables as two reais: the real part and then the Imaginary part. Arrays have an entry for each element of the array, and the displacement in actual memory locations of each of its elements with respect to the start address of the array IS given in the ADDRESS field of its INITIALIST record entry. The real address for the elements initialized is not entered until procedure FILL_ADDRESS_INITIALIST is called after storage allocation has occurred. This will just add the address in the symbol table to what is already in the ADDRESS field in an INITIALIST entry. Types of the initialized variables and dimensions of the arrays whose elements are being initialized must have been completely defined before the initialization specifications.

9.2 The DATA statement

Procedure DATA-STMT scans and processes a DATA statement and builds the list of the variables to be initialized.

A DATA statement is composed of the alternate appearances of a variable list followed by the initialization constants enclosed by the slashes. Procedure FORM_VAR_LIST processes a variable list and adds nodes to the initialization list for the variables to be initialized. Procedure FILL-VALUES then processes the upcoming list of constants and updates the list with the initial values in the nodes for the variables just inserted. Variable FIRST-IN-LIST is returned from FORMVARLIST pointing to the first element of the group just inserted and is used by FILL-VALUES to teli where to start entering the initializatron values.

Here is a more detailed description of the procedures used:

Procedure FORMVARLIST gets and inserts the names of the variables to be initialized into the symbol table, indicating that they are being initialized by setting the field INITIALIZED to TRUE. It then creates the entries in the initialization list for these variables by calling procedure EXTEND-LIST.

Since the variable list can consist of arbitrarily nested loops, FORMVARLIST uses special data structures and an recursive algorithm to process the variable list. These are presented in the next section.

Procedure EXTEND-LIST does the actual building of the initialization list. The information inserted by this routine consists of a pointer to the symbol table entry for the element being initialized, its displacement in memory with respect to the beginning of the array, which is 0 for a simple variabie, the size of the location and the flag CONTINUING which is used to indicate if the current location is a continuation of the location in the previous node, as in the succeeding elements in the initialization of whole arrays and the second halves of complex variables.

Procedure FILL-VALUES updates the list of variables in the initializatron list with the corresponding initial values. FIRST-IN-LIST points to the first element of the list that needs an initialization value and POINT-TO-LIST is used to traverse the list of INITIALIST records while saving the values in the AMOUNT field. For each initializatron value, this procedure gets the number of times the value is repeated. INSERT_VALUE is then called this number of times. Fieids NEGATIVE and STRLEN of INITIALIST are set directly in FILL-VALUES depending on the type of the constan t. For string constants, INSERT_VALUE is called as many times as required depending on the length of the string, and depending on the flag CONTINUING.

Procedure INSERT-VALUE completes the information in the INITIALIST record entry by filling in the lexeme type and the initialization values expressed as an array of characters.

The procedures EXTEND-LIST, FILL-VALUES and INSERT_VALUE are also used in processing initializations in type-specific declaration statements.

9.3 Procedure FORMVARLIST

In order to handle arbitrarily nested loops in a variable list in the DATA statement, this procedure uses two phases to process a variable list. The first phase, represented by procedure CONSTRUCT, builds a list recursively according to the loop structure in the variabie list. The second phase, represented by procedure EXTEND, traverses the list just created recursively and, in the process, expands the nested loops into linear counts of initializations being added at the end of the global initialization list.

The list constructed is made up of two kinds of records, which represent respectively an element in a variable list and a loop. The structures of these two records are as follows:

```
( • NOOES THAT FORM A LIST OF VARIABLES, ANY OF WHICH CAN INSTEAD BE
   ANOTHER LEVEL Of AN IMPLIED LOOP, IN WHICH CASE ISLOOP IS TRUE •   )
VARREC = RECORD
           NEXT: ↑VARREC;                (* NEXT IN LIST *)
           CASE ISLOOP: BOOLEAN OF
           TRUE: ( NEXTLEVEL: POINTLEVELREC); (* POINT TO THE NOOE
                                  THAT REPRESENTS THE NESTED LEVEL *)
           FALSE: (SPTR: POINTSYMBOL;
                   NUMSUBS: INTEGER;        (* # Of SUBSCRIPTS *)
                   SUBSINFO: ARRYSUBSCRIPTS);
           END;

(* NOOE TO REPRESENT A LEVEL OF IMPLIED LOOP. VARLIST POINTS TO THE
   LIST OF VARIABLES (OR AOOITIONAL NESTED LOOPS) THAT BELONGS TO THIS
   LEVEL.  PREVIOUS POINTS TO THE NODE OF THE LEVEL INSIDE WHICH THIS
   LOOP IS NESTED.  CONTROLVAR KEEPS THE NAME OF THE CONTROL VARIABLE
   OF THIS LOOP.  CURRENTVAL IS USED TO STEPS FROM STARTVAL TO
   ENDVAL INSIDE PROCEDURE CONSTRUCT. •   )
LEVELREC = RECORO
             VARLIST: POINTVARREC;
             CONTROLVAR: THENAME;
             STARTVAL,ENDVAL,STEPVAL,CURRENTVAL: INTEGER;
             PREVIOUS : ↑LEVELREC;
            END;
```

The recursive algorithm to process a variable list is then as follows:

Formvarlist:

1. Call CONSTRUCT to scan and build the list representation for the variable list.

2. Call EXTEND to do the extensions to the initialization list according to the structure just created.

Construct:

1. While not end of variable list,
   (a) Create a VARREC node.
   (b) If next item is a loop, current node is a loop. Create a LEVELREC node pointed to from the VARREC node.
      (1) Enter the loop information to the LEVELREC node.
      (2) Call CONSTRUCT to scan and build the list representation for the variable list pointed to from the LEVELREC node.
   (c) Else next item is a variable. Enter the variable information together with any subscript specification in the VARREC node.
   (d) Append the VARREC node to the end of the list being built.

Extend:

1. For each node in the VARREC list do:
   (a) If current VARREC node is a loop, get to the LEVELREC node.
      (1) Initialize CURRENTVAL to STARTVAL.
      (2) While CURRENTVAL ≤ STEPVAL,
         a. Call EXTEND using the variable list of this loop.
         b. Increment CURRENTVAL by the amount given by STEPVAL.
   (b) Else current VARREC node is a variable. Do the extension to the initialization list for this variable, array element or whole array. If any subscript is an identifier, the value of the subscript is given by the CURRENTVAL field of the LEVELREC node in which the subscript identifier is the control variable.

9.4 Procedure FILL_ADDRESS_INITIALIST

This procedure finds the address of **a** variable once storage has been allocated to it and enters the address in its INITIALIST entry, The procedure is called after STORAGE-ALLOCATION has been called, which occurs after processing the last declarative statement and before the first statement function or executable statement in a program unit.

Global variable NEXTININIT is used to remember the record entry of the last variable initialized for the previous program unit. Ail the entries in INITIALIST after that entry **are** traversed and the corresponding addresses are entered.

The displacement information, stored in field ADDRESS, is computed by adding the value already in the ADDRESS field of INITIALIST and the address stored in the symbol table entry for the variable. This IS because the distance of an array element from the start address of the array was previously stored here. If it is a simple variable, this ADDRESS field would have previously stored 0. Field LEVEL is obtained directly from the LEVEL field in the symboi table entry. After these two pieces of information are obtained, the pointer to the symbol table entry is set to NIL, so that when the symbol table is cleared at the end of the current program unit, no pointer points to its entries and the space used by the symbol table can be reclaimed for other uses.

At the end, NEXTININIT is updated to point to the last element of the initialization list that corresponds to the last variable initialized in the most recently compiled program unit.

9.5 Procedure VARINITIALIZATION

This procedure is called by the main procedure after ail the program units **are** compiled. It generates code for the initialization of variables and the loading *of* FORMAT specifications into memory at execution time, the latter being done by calling procedure INIT_FORMATS (see Section **25.2).**

The code for the initiaiization of variables is placed inside a special U-Code procedure, created for the compiler, called $INIXX. A call to procedure $INIXX is always executed before anything else in the compiled U-Code program.

The head of the special procedure $INIXX is generated by calling procedure BLKCODE_GENERATION. Then, code for the body of procedure $INIXX is generated. This consists of a series of LDC-STR U-Code instructions that load the constant values on the stack and store them into the variables' locations in memory. String constants are loaded into variable addresses using the LCA-LDA-MOV sequence of U-Code instructions.

10. Storage Allocation Structure

In U-Code, as in P-Code, there are a number of static *levels*, each of which may have one or more procedures associated with it. Each procedure owns a set of local variables. When a procedure is entered, space for its variables is allocated. On exit, the space is deallocated. Thus, the values of all the local variables of a procedure are undefined when that procedure is entered.

In common Fortran implementation, however, ail of the variables of each subroutrne are own variables; that is, their values remain the same between the end of one invocation of a subroutine and the beginning of the next. Hence, space for ail of these variables must have been allocated at the beginning of program execution, even though some of them may only be accessed when certain subroutines are entered. In U-Code terms, this means that ail variables in a Fortran program must be on some level that is lower than or the same as the level of the main program.

If both the common and regular variables are on the same level, the address of any variable following those declared to be in a common area cannot be definitely determined until the size of that common area is known. To solve this problem, the size of each common area, except the biank common, is restricted to the space that it occupies the first time it is declared in a program unit. The fixed space can be explicitly set using the CSIZ option. The size of the blank common area is unrestricted by assigning to it its own storage level. A storage level is assigned to the rest of the common areas.

Another level is assigned for the storage of the local variables of the program units. In addition, space is allocated in this level for storing (a) the results of expressions, constants or subprogram names when they are arguments in subprogram calls, (b) format strings and (c) parameter addresses for parameters to multiple entry subprograms.

The levels in the U-Code generated by UFORT are distributed as follows:

Level 1 -- non-common variables (dummy procedure)

Level 2 -- ail other common areas (dummy procedure)

Levei 3 -- the blank common area (dummy procedure)

Level 4 -- main block and subprograms

Level 5 -- ail statement functions

The storage for parameter addresses and return values in subprograms and statement functions, together with any temporary location used by the compiler inside their procedures, is allocated in their respective level 4 or 5 stack frames.

Level 5 is used for statement functions because they can only be called from the level 4 subprograms in which they are defined.

U-Code does not require that procedures be in any specific order. Thus, the code for the procedures in levels 1 to 3, which includes how much storage is needed for these procedures, could come after the code for levels 4 through 5. The executions of these three procedures involve only the calls to the procedure of the next higher level.

Here is a Pascal representation of the idea:

1 0.7 Pascal representation

```
program FORVARS;
  var i: array [1..10] of integer; (* variables in the blank common *)

   procedure GENCOMMON;
     var n: array [1. .1000] of integer:
        (* variables in all other commons *)

     procedure BLANKCOMMON;
       var k: real;
          (* all variables not in COMMON areas stored here *)

         procedure USERSUBROUTINE;

           function STATEMENTFUNCTION (real X);

             begin
             STATEMENTFUNCTION : = 2*X;
             end:

           begin (* USERSUBRGUTINE *)
           k := 2.0; (* normal variable *)
           i[1] := 0; (* variable in blank common *)
           end:

         begin (* Fortran main prog *)
         k := 0; (* normal variable *)
         USERSUBROUTINE;
         i[1] := 0; (* in blank common *)
         j[1] := 0; (* in common 1 *)
         end:

    begin (* dummy for general common area *)
    BLANKCOMMON;
    end:

  beg i n (* dummy for b l ank common area *)
  GENCGMMON;
  end.
```

11. Storage Allocation

In the storage allocation process, each variable is assigned a level number and an offset. Procedure STORAGE-ALLOCATION assigns memory locations to the variables declared during the declaration part of a block. The procedure is called after all declarations have been processed and before any statement function declaration or executable statement occurs. Any other variable that appears later in the program without having been previously declared is allocated through procedure SIMPLE_STORAGE, which is called by FSYMBOL. The storage allocation for dummy arguments in subroutines, functions and ENTRY statements are performed in the parameter processing procedures. (See Section 17.2.) The storage allocation for temporaries generated by UFORT is done in the temporary storage management routines. (See Section 13.)

The storage already allocated in the different levels are monitored by *displacement variables* which indicate at the same time the next address available for assignment. The global variable DISPLACEMENT and DISPL GENCOMMON are used for the levels of the non-common variables and general common areas respectively. Variable MAXDISPL_BLANKCOMMON indicates the highest address so far allocated in the level for blank common variables. Every time a space for a variable is needed, the corresponding displacement variable is adjusted, if necessary, to lie on a half, single or double word boundary according to the size of the variable. Its value is then stored in the field ADDRESS of the symbol table. It is then incremented by the proper amount.

The allocation of space is done in a specific order:

1. Common variables and variables equivalenced to common areas. The common areas are allocated in lexicographical order. Inside each area, the variables are allocated in the order in which they were declared as part of the common area. The variables equivalenced to one in the common area are allocated according to the desired equivalence relation.

2. Equivalenced variables with no common element in the equivalence group.

3. All other variables, in lexicographical order.

All common areas, equivalenced variables within a common area and other equivalenced variables begin at a double-word boundary. For the rest of the variables, quarter-word variables begin at the next quarter-word boundary, half-word variables at the next half-word boundary, single-word variables at the next single-word boundary and double- and quadruple-word (complex) variables at the next double-word boundary.

Common variables are passed to procedure STORAGE_ALLOCATION in the form of a list {see Section 7). The list of variables in a common area is pointed to from the PTRCOMLIST field of its common name table entry. The equivaienced variables are represented as a global list of equivalence groups (see Section 7).

Here is a more complete description of how storage allocation is done:

1 I. t Preprocessing equivalence groups

Before any space is allocated, the offsets of the equivalenced variables with respect to the *leader* of the group (the first variable declared in the group) is computed This is done in procedure EQUIV_OFFSETS. It also merges two equivalence groups if a variable is equivalenced in both of them, checking for any index conflict in array elements (e.g. "EQUIVALENCE (A (3) , B (2)) , (A (2) ,B (3) ,C)"). The algorithm used in the computation of the offsets is as described in [Gri7 1].

Procedure MERGE is called by EQUIV_OFFSETS if a variable is equivalenced two times. First, it finds the two entries of the variable in the list of equivalence groups. If the variable appears two times in the same equivalence group, the second one is deleted. If the variable appears in two different groups, the first group is deleted and appended to the beginning of the second one. In this second group, the variables that have already been processed at the moment the double equivalence is found have their offsets adjusted in accordance to the new leader of the group. The doubly equivalenced variable is skipped in the second list and the variables not yet processed will still be at the end of the enlarged group being processed.

1 1.2 Allocating space for common areas

Once all the offsets for the equivalenced variables have been computed and all necessary mergings have been performed, space for the common variables is allocated. The address where the common area begins is given in the STADDR field in the common name table. It is -1 if no space has been allocated for that area in any previously compiled program unit, and in this case, STADDR is set to the next available address in the general common area. If space has already been allocated for the common area, STADDR gives the address where the area was previously allocated, For the blank common variables, allocation always starts with the first address in the level for the blank common area.

If a common variable is also equivalenced, procedure CHECK_EXTENSION is called. This checks for invalid extensions to the left of a common area due to the equivalence, and then assigns addresses to the variables in the equivalence group by calling procedure ALLOC_COMMON_AND_EQUIV. After space is allocated for all the common variables of an area, extensions to the right of the common area are checked. See Section 7.5 regarding how the initial length of a common area is determined.

1 1.3 Allocating space for non-common variables

Once space has been allocated for all the common variables, the list of equivalence groups is traversed and space is assigned to those groups not yet processed. Finally, the symbol table is traversed in alphabetical order and space for all remaining variables is allocated.

12. U-Code generating routines

Almost all code that is written in the U-Code file is generated by one of the U-Code generating routines. There are a few cases in which U-Code is written directly using WRITELN.

The U-Code generating routines are made to cope with the syntax of U-Code instructron types. The three routines GEN, GEN2 and GEN3 cover most of the general U-Code instructions. The rest of the routines generate special U-Code instructions or groups of instructions.

The parameters to the U-Code generating routines convey the field contents of the instruction to be generated. The most common fields are the U-Code operand type, memory type, block number, address and location size. The U-Code operand type together with the location size is conveyed by a single type parameter, of type POINTDEFTYPE (see Section 8.1). The compiler processes addresses in units of half-words. Currently bit addresses are used, so that all address parameters have to be multiplied by the constant BYTELEN (the number of bits per byte) before written out. Since the symboi table keeps only the level information of the variables, the block number is given as the parameter by indexing into the global array SEGLEN using the level as index. The array SEGLEV is updated whenever a new U-Code block is entered.

The LDC instruction is generated by a number of different procedures distinguished by the forms in which the constants are passed to the procedures:

GENLOADNUM - the constant is to be taken directly from the Fortran statement kept in the array LEXSTRING. The pointer to the lexeme is passed.

GENLDC - the constant is passed as a string of 20 characters which can contain any possible double precision number.

GENLOADINT, GENLOADBOOL, GENLOADCHAR - the constant is passed in integer, boolean and character forms respectively.

GENOREAL - the constant is always the floating point zero.

Other U-Code generating routines are:

GENLOADSTRING - given a pointer to a string lexeme, generates code to load that lexeme.

GENLABEL - prints a U-Code label definition, e.g. "L15 LAB".

GENDEF, GENCLAB, GENLDA, GENXJP, GENCSP, GENMST, GENCUP, GENEND, GENLDP, GENENT - generates the given instruction.

GENSEGCODE - generates the dummy blocks (see Section 10).

GENLEXES - generates the LEX instruction at the beginning of each U-Code block according to the global array SEGLEV.

The following two procedures are called from the above U-Code generating procedures:

PRINTLABEL - prints a U-Code label, e.g. "L15".

PRINTNAME - prints the name of a program unit in U-Code form, e.g. "PEPE0003". The maximum iength of the name is 5 letters. The maximum segment number is 999. Each procedure has its own segment number. The global variable **SEGNUMBER** always contains the segment number that was last allotted.

13. Temporary storage management

Temporary locations are used in UFORT in a number of places. They are made available for reuse whenever possible. New temporary locations are generated only if the existing ones are not free. Temporary locations are used in the foilowing cases:

1. In processing complex number arithmetic.
2. In different cases connected with complex numbers: the assrgnment to a complex variable with an indirect address, the relational and bitwise operations on complex operands and the printing of a complex number by the **PRINT** statement processor.
3. In processing the assigned GOTO statement.
4. In processing the arithmetic IF statement.
5. In processing READ and WRITE of whole arrays.
6. In **DO** statements when the final value or step value is an expression (the temporary locations for these cannot be reused).
7. In connection with type coercions and error recovery inside ARITH.
8. In generating in-line code for some intrinsic or standard functions.

Temporary locations are allocated in the level of the program unit being compiled, and thus they exist only while the program unit is being executed. UFORT distinguishes between two memory types in U-Code: type R (registers) and type M (*main* memory). It assumes that each U-Code procedure has a number of registers available for its local storage. The constant MAXREGS defines this number. In addition, the constant MAXPREGS defines the maximum number of registers that can be allocated to the parameters of a program unit. Temporary locations are allocated in type M memory only after no more R memory is available. Since some temporary locations are used in connection with loops, and temporary locations are reused whenever possible, this scheme contributes to greater efficiency when the U-Code are executed.

The two temporary storage management procedures are GETTEMP and RELTEMP. GETTEMP gets a temporary location and returns its level, address and memory type. RELTEMP is called to specify an allocated temporary location being now available for reuse somewhere else as a temporary storage location.

The temporary locations are kept in a linked list pointed to by global variable **TEMPLOCHEAD.** In the beginning, the list contains no node. The list is lengthened as more and more temporary locations are allocated. The order of each node in the list corresponds to the order in which they are allocated. The structure of each node is:

```
TEMPLOCNOOE  =  RECORD
                  LOC,
                  SIZE:  INTEGER;
                  MTYPE:  CHAR;
                  FREE:  BOOLEAN;
                  NEXT:  ↑TEMPLOCNODE;
                END;
```

GETTEMP first searches the list to see if there is a temporary location of the appropriate size that has already been claimed as a temporary location but is now free. The search starts from the beginning of the list, so that any type R memory location is found first. If there is none, it claims a new one by incrementing the displacement variable of the appropriate level and memory type by an amount which is the size of the location needed plus any extra it needs to assure that the location starts on a single-word boundary. The new node to remember this temporary location is added to the list.

RELTEMP merely searches through the list until it finds the specified location, then sets FREE to TRUE.

TEMPLOCHEAD is reset to NIL before the start of a new program unit or statement function, since the temporary locations previously allocated no longer apply.

14. Loading and storing variables

The procedures used to generate code to load and store. variables are LOAD__VAR, LOAD__VAR__ADDR, LOAD-ARRAY-ELEMENT and STORE__VAR. To load the value of a variable, LOAD__VAR is called. To store a value in a variable, STORE VAR is called, then the value is loaded (usually by ARITH) and then STOREVAR is called. Complex variables are handled differently inside LOADVAR and STOREVAR as each variable requires the loading or storing to be performed twice.

Variables are accessed differently as to whether it is a regular variable, a variable passed as a parameter or an array element. For the last two cases, it is necessary to access the variables indirectly by loading the address on the stack first, and then doing a load or store Indirect. The loading of the address is done by LOAD__VAR__ADDR.

LOAD__VAR__ADDR is passed a pointer to the symbol table for the variable in question. If the variable is not an array variabie, it loads its address. If the variable is an array, it loads its address, and then calls LOAD-ARRAY-ELEMENT, which reads the subscripts and generates code to calculate the offset.

The offset of an array element is computed by a loop which iterates according to the number of subscripts specified. For an array A of dimensions (bl , $b_2, \dots, b_n$), the offset for the element $A(i_1, i_2, \dots, i_n)$ is given by:

$$i_1-1+(i_2-1+(i_3-1+(\dots(i_{n-1}-1+(i_n-1)*b_{n-1})*\dots)*b_3)*b_2)*b_1$$

If the first m dimensions of the array have constant bounds, the above algorithm can be made more efficient by accumulating the decrements-by-l of the '2nd to $(m+1)$th subscripts into one single Offset adjustment. As an illustration, suppose the array A above has all constant dimensions. Then the offset computation can be compressed into:

$$i_1+(i_2+(i_3+\dots(i_{n-1}+i_n*b_{n-1})*\dots)*b_3)*b_2)*b_1-(((\dots((b_{n-1}+1)*b_{n-2}+1)*\dots)+1)*b_1+1)$$

The last adjustment term is computed during compile time when processing the dimension declaration of the array.

In the following example, the array has both constant and variable dimensions.

14.1 Example of indirect load and store

```
Fortran:     SUBROUTIHE X (I)
             DIMENSION J(3,4,I)
             J(2,3,5) = I
             RETURN
             END

U-Code:
           X0000076   EMT   P 4  76 1 0
            LEX      1  1
            LEX      2  72
            LEX      3  73
            PSTR  A  R  76 0  36
            LDA      M 1  504  0     ;load address of array J
            LOC  J  36  2
            LDC  J  36  3
            LDC  J  36  5
            LDC  J  36  4
            MPY  J
            ADO  J
            LDC  J  36  3
            MPY  J
            ADO  J
            DEC  J  16
            IXA  J  36                ;up to here, load address of J(2,3,4)
            LOO  A  R  76  0  36      ;load address stored at address of I
            IL00  J  0  36            ;load content of address just loaded
            ISTR  J  0  36            ;store value at address 2nd on stack
            RET                       ;this is from the RETURN statement
            RET                       ;this is always generated
            DEF  R  36
            DEF  M  72
            END  X0000076
```

15. Expression Evaluation

Expression evaluation is done by recursive descent. Although this is somewhat less efficient than using operator precedence, it is cleaner and makes it easier to deal with parentheses.

Expression evaluation procedures are divided into logical expression procedures and arithmetic expression procedures. Logical expressions are expressions involving logical operators, such as .AND.. They may include arithmetic expressions if relational operators, such as .EQ., occur inside the logical expression. Arithmetic expressions are constants, variables, function calls or other arithmetic expressions connected by arithmetic operators. If the logical operators .AND., .OR. and .NOT. are used in arithmetic expressions, the respective bitwise operations on the operands are implied.

Since the type of an expression may not be known until after the expression has been compiled, as in the case of an expression which is the parameter in a function call, the compilation is always started by calling the highest level logical expression procedure, called ARITH. ARITH expects the global lexeme pointer LXC to be pornting to the beginning of the expression when it is called, and leaves it pointing to the lexeme after the expression. All the intermediate parsing procedures return the data type of the parts of the expression which they parse to their next higher level calling procedure, and ARITH returns the data type that will be left on the top of the stack when the whole expression is evaiuated.

Bitwise operations are done in U-Code using the set operations, with .OR. corresponding to set union (UNI) and .AND. corresponding to set Intersection (INT). The .NOT. operation is handled using the set difference operation (DIF) between a full word of 1's and the .NOT. operand.

15.1 Syntax

The syntax for expressions is as follows:

```
logical-expression ::= logical-term {".OR." logical_term}

logical-term ::= logical-factor {".AND." logical_factor}

logical-factor ::= {".NOT."} relational-expression

relational-expression ::= arith-expr rel-operator arith-expr

rel_operator ::= ".LE." | ".LT." | ".GE." | ".GT." | ".NE." | ".EQ." |
                 "<" | ">" | "="

arith-expr ::= term {addop term)

term ::= {addop} factor {multop factor}

factor ::= {primary} {"**" primary}

addop ::= "+" | "-"

multop ::= "*" | "/"

primary ::= "(" arith-expr ")" | integer-constant | real-constnat |
            complex-constant | logical-constant | variable | array-element |
            function-call
complex-constant ::= "(" arith_expr "," arith-expr ")"
logical-constant ::= ".TRUE." | ".FALSE."
```

I 5.2 *Processing identifiers*

When ARITH encounters an identifier, it must determine wh ether it is a variable, a call to a stand ard function, a call to a u ser-defined function or a call using a function dummy argument.

There are two procedures for processing function calls: STANDARDFUNC, which processes calls to intrinsic and standard external functions, and USERFUNC, which processes calls to statement . functions and external functions. For the latter, refer to Section 19.

One of the fields of every record in the symbol table is S_FUNCSUBR. It has one of the following values:

```
FUNCTYPE = (NOTEXTERNAL,EXTERNAL,EXTSUBR, EXTFUNC,STMTFUNC, INTRINSTDEXT,
            PARAHPROC ) ;
```

How a symbol functions in the program is determined by its FUNCTYPE attribute:

NOTEXTERNAL denotes that the identifier is a variable or array name, or the value for this field has not yet been asserted;

EXTERNAL means the identifier has been declared in an EXTERNAL statement but cannot yet be classified as EXTSUBR, EXTFUNC or PARAMPROC;

EXTSUBR, EXTFUNC, STMTFUNC, INTRINSTDEXT, PARAMPROC denote an external procedure. an external function, a statement function, an intrinsic or standard function and a procedure parameter respectively.

This is the way ARITH processes symbois:

1. Look it up in symbol table. This means that if the symbol is not already there, it is entered, with, among other things, the S_FUNCSUBR field set to NOTEXTERNAL If it has appeared in this program unit before, then S_FUNCSUBR will already contain the needed information.

2. If we already know it is a user function, then call USERFUNC.

3. Else if we already know it is an intrinsic or standard function, then call STANDARDFUNC.

4. Else if next lexeme is not a left parenthesis or it has been dimensioned, then it must be a simple variable or array element; call LOAD_VAR (see Section 14).

5. Else if it is a dummy argument, it must be a function parameter; call procedure USERFUNC to process the call.

6. Else if it is in the standard function table, set S_FUNCSUBR to INTRINSTDEXT to indicate that it is a standard function and call STANDARDFUNC.

7. Else it must be a user-defined subprogram; set S_FUNCSUBR to EXTFUNC to indicate this, then enter it in the EXTERNAL table and call USERFUNC.

15.3 Type checking and error recovery inside ARITH

UFORT conducts full type-checking and always emits explicit conversion code whenever type coercions are required. This eliminates the need to look out for implicit type conversions in any translator or interpretor of U-Code generated from UFORT.

The checks for type compatibility involving expressions are done using the procedures MATCHTYPE or FITTYPE, which are called on different occasions. MATCHTYPE is used when the types of two values are to be matched, performing coercion on one of them if necessary. Coercions are always done in the direction of integer values to real values to complex values. For example, if one of the values is a real and the other is a complex, the real value is converted to its corresponding complex number, and not the other way round.

FITTYPE is used when the type of a value is to be fitted to a desired result type, as in the case of an assignment to a variabie. In this case, any coercion performed will be the conversion of the value to the result type.

An additional procedure, MATCHSIZE, is called from both MATCHTYPE and FITTYPE. it is for checking the correspondence of sizes after the types have been matched. If size incompatibility occurs, the CVT or CVT2 instructions will be generated for size coercions, with warnings output at the same time.

UFORT always attempts to generate correct U-Code even if an error occurs. In the case of an arithmetic expression, the fix-up of the generated U-Code and exit from the nested parsing procedures are effected in the following manner. Each parsing procedure assumes no error occurs in the procedures which it calls for parsing its subexpressions, and if it discovers an error itself, it will finish parsing at the earliest possibility, generating any dummy instructions which it is expected to generate in normal processing. Thus, a call to ARITH will always finish with a single result on top of the stack. Since the global error message routine only outputs one error message for each statement, the error message output is appropriately that from the parsing procedure that first discovers an error.

15.4 Example

```
Fortran:  IF (3.2 * I .EQ. 5.1 ** 3) GOTO 233

U-Code :
          LDC  R 36 3.2
          LOD  J M 1 504 36     ;load value of variable I
          CVT  R J              ;float value of I
          MPY  R
          MST   4
          LDC  R 36 5.1
          PAR  R M 0 0 36
          LDC  J 36 3
          PAR  J M 0 0 36
          CUP  R 52 RIEXP052 2 1 ;call exponentiation library function
          EQU  R
          FJP  L1001
          UJP  L1002
```

15.5 The assignment statement

The assignment statement works as follows:

It first looks up the symbol in the symbol table and calls LOAD_VAR_ADDR to load the address on the stack, if necessary. It sets the global iexeme pointer, LXC, to point to the lexeme after the equal sign. It then calls ARITH to evaluate the expression, followed by ASSIGNVALUE to do the assignment.

ASSIGNVALUE checks whether the expression is a string or not. If not, STOREVAR is called (see Section 14). Otherwise, it calls STORESTRING.

STORESTRING is used to store a s&ring into any kind of variabte. It generates code to load the string into the address indicated using the MOV instruction. If the string is larger than the size of the variable, the extra characters are Ignored. If the string is shorter, the variable is padded with the null character.

16. Complex Number Arithmetic

Complex numbers are loaded on the U-Code stack as two real values, with the real part second and the imaginary part on top on the stack. Since there is no U-Code instructions that takes a pair of stack values as an operand, an operation on complex numbers consists of composite U-Code instructions. The SWP and DUP instructions are used extensively. Storing values into temporary locations and loading them back later are necessary.

Each complex number operation finishes with the complex result on top of the stack. If the complex result is to be combined again with another complex operand, greater efficiency can be achieved if one part of the previous complex result is ieft in its temporary location. But this then involves greater complexity in the processing algorithm, and so is not pursued.

The methods implemented use the least number of temporary locations and also the least number of toad and store instructions, although they certainly do not generate the least number of U-Code instructions or try to minimize the height of the U-Code stack.

In the following, the methods for complex number arithmetic are illustrated with examples. Note that some manipulations on the first operand are performed on seeing the operator and before processing the second operand. In the description, the two complex operands will be referred to as (X1,Y1) and (X2,Y2) respectively.

16.1 Addition and subtraction

Fortran: c = C1 + c2

U-Code:

```
        LOD  R M 1 576 36      ;load X1
        LOD  R M 1 612 36      ;load Y1
        SWP  R R               ;swap X1 and Y1
        LOD  R M 1 648 36      ;load X2
        LOD  R M 1 684 36      ;load Y2
        STR  R M 74 72 36      ;store Y2 temporarily
        ADD  R                 ;X1 + x2
        SWP  R R               ;swap Y1 and (X1 + X2)
        LOD  R M 74 72 36      ; load Y2 back
        ADD  R                 ;Y1 + Y2
        STR  R M 1 540 36      ;store (Y1 + Y2)
        STR  R M 1 504 36      ;store (X1 + X2)
```

Subtraction is similar, and is not repeated.

16.2 Multiplication

Fortran: c = C1 * C2

U-Code :

```
        LOD  R M 1 576 36      ;load X1
        LOD  R M 1 612 36      ;load Y1
        SWP  R R               ;swap X1 and Y1
        NSTR R M 74 72 36      ;store X1 temporarily
        SWP  R R               ;swap Y1 and X1
        DUP  R                 ;duplicate Y1
        LOD  R M 74 72 36      ;load X1 back
        LOD  R M 1 648 36      ;load X2
        LOD  R M 1 664 36      ;load Y2
        STR  R M 74 108 36     ;store Y2 temporarily
```

```
NSTR R M 74 72 36     ;store X2 temporarily
MPY  R                ;X1 * x2
SUP  R R              ;swap Y1 and (XI * X2)
LOO  R M 74 108 36    ;load Y2 back
MPY  R                ;Y1 * Y2
SUB  R                ;(X1 * x2) - (Y1 • Y2)
SWP  R R              ;swap Y1 and ((XI • X2) - (Y1 • Y2))
LOO  R M 74 72 36     ;load X2 back
MPY  R                ;Y1 * x2
STR  R M 74 72 36     ;store (Y1 * X2) temporarily
SWP  R R              ;swap X1 and ((XI * X2) - (Y1 * Y2))
LOO  R M 74 108 36    ;load Y2 back
MPY  R                ;X1 * Y2
LOO  R M 74 72 36     ;load (Y1 * X2) back
ADO  R                ;(X1 • Y2) + (Y1 * X2)
STR  R M 1 540 36     ;store imaginary part of result
STR  R M 1 504 36     ;store real part of result
```

16.3 Division

Fortrsn: C = Cl / C2

U-Code:

```
LOO  R M 1 576 36     ;load Xl
LOO  R M 1 612 36     ;load Yl
SWP  R R              ;swap Xl and Yl
NSTR R M 74 72 36     ;store Xl temporarily
SWP  R R              ;swap Y1 and X1
OUP  R                ;duplicate Y1
LOO  R M 74 72 36     ;load back Xl
LOO  R M 1 648 36     ;load X2
LOO  R M 1 684 36     ;load Y2
STR  R M 74 108 36    ;store Y2 temporarily
NSTR R M 74 72 36     ;store X2 temporarily
MPY  R                ;X1 * x2
SWP  R R              ;swap Y1 and (Xl * X2)
LOO  R M 74 108 38    ;load Y2 back
MPY  R                ;Y1 * Y2
ADO  R                ;(X1 * X2) + (Y1 * Y2)
LOO  R M 74 72 36     ;load X2 back
SQR  R                ;X2**2
LOO  R M 74 108 36    ;load Y2 back
SQR  R                ;Y2**2
ADO  R                ;X2**2 + Y2**2
NSTR R M 74 144 36    ;store (X2**2 + Y2**2) temporarily
DIV  R                ;((X1 * X2) + (Y1 * Y2)) / (X2**2 + Y2**2)
SWP  R R              ;swap Y1 and real part of final result
LOO  R M 74 72 36     ;load X2 back
MPY  R                ;Y1 * x2
STR  R M 74 72 36     ;store (Y1 * X2) temporarily
SWP  R R              ;swap Xl and real part of final result
LOO  R M 74 108 36    ;load Y2 back
MPY  R                ;X1 • Y2
NE6  R                ;- (Xl * Y2)
LOO  R M 74 72 36     ;load (Y1 • X2) back
ADO  R                ;(Y1 * x2) - (Xl • Y2)
LOO  R M 74 144 36    ;load (X2**2 + Y2**2) back
OIV  R                ;((Y1 * x2) - (Xl * Y2)) / (X2**2 + Y2**2)
STR  R M 1 540 36     ;store imaginary part of final result
STR  R M 1 504 36     ;store real part of final result
```

16.4 Complex-valued functions

Since U-Code does not have multiple return values for functions, complex-valued functions in Fortran are compiled into U-Code functions that return the addresses of their complex results. The responsibility of loading the complex result of a function on the stack then rests on the callee. The following illustrates how a callee does the call to a complex-valued function:

```
Fortran:    C = CFUNC(C1)

U-Code:
            MST   4
            LDA     M 1 576 72          ;load address of Cl
            PAR   A M 0 0 36
            CUP   A 76 CFUNC076 1 1     ;call to complex-valued function
            OUP   A                     ;duplicate address returned
            IL00  R 0 36                ;load real part
            SWP  R  A
            IL00  R 36 36               ;load imaginary part
            STR   R M 1 540 36          ;store imaginary part
            STR   R M 1 504 36          ;store real part
```

17. Subroutine and Function Statements

Procedures SUBR_STMT and FUNC_STMT process the subroutine and function statements. Both of them initiate a new program unit by calling procedure INITBLOCK The global flag IN_SUBR_FUNC is set to TRUE whenever the compiler is processing a subprogram, and the global pointer SEGPTR points to the symbol table entry of the subprogram name. All the parameters of a function or a subroutine are passed by reference, thus each is allocated 4 quarter-words of storage (the space required for an address).

Whenever an identifier used as a variable is encountered in the executable part of a program unit, its STYPE field in the symbol table entry is checked, and either the FUNCTYPE field is NOTEXTERNAL or the symbol table entry is identical to that pointed to by SEGPTR, in which case it is the function variable. An identifier not satisfying these conditions cannot be used as a variable in that program unit.

The fields ADDRESS, S-EXPLICIT, USED-RHS and USED-LHS of the symbol table entry of a subroutine are not used. Its STYPE field has to be set to NONE so that its use as a variable does not pass the above test. The used and *defined* information for functions and subroutines is kept in the external table instead.

17. 1 Initialization of a segment block

The initialization of the global variables when a new block is found is done by procedure INITBLOCK. This procedure performs the following steps:

1. It clears the symboi and label tables, the list of equivalenced variabies and the list of DO's that are still open.

2. It restores the standard default values for variables not declared by modifying IMPLIARRAY.

3. In the common table, it sets the fieid PTRCOMLIST for each area to NIL, since the compiler is ready to build a new list of common variables for the common area in the next program unit. COMMONSIZ, the variable in charge of the CSIZ option, is also reset to 0.

4. It sets to FALSE the global variables AFTER_STORAGE-ALLOCATION, which indicates if the storage allocation of the variables declared in the program unit has occurred, and HAS-RETURN, which indicates if a RETURN statement for the program unit has been encountered.

5. It reinitializes the displacement pointers for the level of the program unit.

6. It initializes the global variabie IFDEST, to indicate that no logical IF statement is being processed.

17.2 Processing dummy arguments

Procedure DUMMY-PROCESSING scans the parameters of a subroutine or a function. allocating space for them and inserting their names, levels (always 4), addresses, and an indication that they are dummy arguments in the symbol table.

In allocating space for the dummy arguments in the level of the program unit, two memory types, type R (registers) and M (main memory) are available. The maximum number of type R memory available for parameters is set by the constant **MAXPREGS**. If the number of dummy arguments exceeds **MAXPREGS**, the remaining parameters are allocated in type M memory. Unused space of type R within the range specified by **MAXPREGS** is available for use as temporary locations. The constant **MAXPREGS** is never greater than **MAXREGS** (See Section 13).

Eight quarter words of type M memory are always reserved starting at address 0 for the return value of a U-Code procedure.

Dummy arguments to muitipie entry subprograms are processed in a different way. See Section 18.

17.3 Subroutine statement

After the call to **INITBLOCK**, routine **SUBR_STMT** inserts the subprogram name in the symbol table with type **NONE** and level 4. The symbol table is updated by a call to **FEXTNAME**. Then it calls the procedure to process the dummy arguments.

17.4 Function statement

Procedure **FUNC_STMT** calls **INITBLOCK** to initialize a new block, gets the type of the function if this is specifically indicated, gets its size modification if specified, inserts the function name in the symbol table indicating its type, size and address (level 4, displacement 0), and processes its dummy arguments by calling procedure **DUMMY-PROCESSING**.

The return value of complex functions are not returned in displacement 0 of the type M memory at level 4 because 2 separate values have to be returned. Instead, space is allocated for it after the space reserved for the function parameters on the level of the function, in type M memory. The address of this space is the return value of the function, and so an indirect reference is needed in order to access the complex returned value of the function. For this reason, such functions are declared internally as being of type *address*.

17.5 Code generation

Code for the head of the new program unit is generated in procedure **BLKCODE_GENERATION**. This procedure is called by global procedure **BLOCK** after all the declarations of the program unit have been processed. This is necessary because all the code for the statement functions must be generated before the code for the head of the program unit is generated.

17.6 Example

```
Fortran: INTEGER FUNCTION X(1)
         X = 2 * I
         RETURN
         END

U-Code:
        X0000076 ENT J 4 76 1 1
         LEX  1  1
         LEX    2 72
         LEX    3 73
         PSTR A R 76 0 36
         LDC J 36 2            ;load constant 2
         LOD A R 76 0 36       ;load address stored at address of I
         ILOD J 0 36           ;fetch content of this address
         MPY  J                ;compute 2* I
         STR J M 76 0 36       ;store at address 0 for the return value
         PLOD J M 76 0 36
         RET                   ;return generated due to the RETURN statement
         PLOD J M 76 0 36
         RET                   ;return generated at the end of all program blocks
         DEF  R  36            ;type R storage
         DEF  M  72            ;type M storage
         END  X0000076
```

18. Multiple Entry Subprograms

Multiple entry subprograms in Fortran provide two features to the Fortran user: (a) a program unit can be entered not just at the beginning of the program block, but at any defined entry point in the program unit; (b) since a call to an entry point involves only the dummy arguments of that entry point, the parameters to the program unit can be set during different calls, remaining intact in the instances that the program unit is not active.

Since multiple entry facilities do not exist in U-Code, UFORT handles the above features by special means. Some restrictions are imposed to enabie UFORT to preserve its one-pass characteristics {See Section 2.8).

18.1 The multiple procedures

A multiple entry subprogram in Fortran is compiled into a number of U-Code procedures, one for each entry point (inciudmg the normal entry point at the beginning of the program unit), plus an extra one which represents the body of the program unit. This will be called the *multientry* procedure while the former ones wrii be called *entry* procedures. Ail these procedures are at the level for program units (level 4).

The entry procedures bear the names of their respective entry statement names, and their parameters are those of their entry statements. Each of these procedures calls the multientry procedure with a single parameter giving it the entry point to branch to.

The multientry procedure always has only the single branch parameter as its dummy argument. It contains the complete code for the body of the multiple entry subprogram, with U-Code labels at the places of the entry statements. In addition, there is a jump table containing jumps to the labels of the various entry points. On entrance to the multientry procedure, the branch parameter is used to determine the jump to the correct entry point.

Since each entry statement has its own U-Code procedure, a call to an entry statement IS just an ordinary procedure call to the corresponding entry procedure. Therefore, calls to entries are processed in the same way as ordinary calls.

Because UFORT is one-pass, it does not know about the entry points of a multiple entry subprogram until after the whole program unrt is processed. Thus, it has to retain the information about the entry procedures and then generates the U-Code for them after the program unit is processed. Also, the jump table has to be put at the end of the multientry procedure since the number of entries is not known until that point.

The entry point identifiers are entered in the external name table, since they are regarded as user-defined subprograms to the rest of the program.

18.2 Global storage of parameter addresses

In order to preserve the identities of actual parameters during the time that the procedure IS not active, the parameter addresses are stored in space specially allocated for the multiple entry subprogram dummy arguments in the global storage level (level 1). During processing of the body of the program unit, the symbol tabie entries of the dummy arguments indicate these addresses.

The addresses of actual parameters are transmitted to the entry procedures in calls to entry points. The entry procedures keep the call parameters in their own storage level (level 4). Before calling the multientry procedure, the entry procedures copy the addresses of the actual parameters to the locations in the global storage level, and in the multientry procedure, the parameters are accessed only through the addresses as stored in the global locations. Each dummy argument has a unique location in the global level, even if it appears in more than one parameter lists, including that of the subprogram heading. If a dummy argument is not involved in a call, the content of its global location is not affected.

Because a dummy argument and a local variabie is accessed in different ways, UFORT has to distinguish between these two types of variables when processing them. Thus, it IS necessary to forbid the appearance of a dummy argument in the program unit before its appearance in a dummy argument list.

18.3 The data structure

The data structure used in processing multiple entry subprograms is solely for the purpose of retaining information for use in generating the jump table and the U-Code entry procedures after the body of the program unit has been compiled. The record types used are defined as follows:

```
(* THIS RECORD REPRESENTS AM ENTRY POINT FOR A MULTIENTRY PROCEDURE.
   ONE IS CREATED FOR THE MULTIENTRY SUBPROGRAM HEADING AND EACH ENTRY
   STATEMENT IN THAT SUBPROGRAM, IN THE OROER Of THEIR APPEARANCES. THIS
   IS USED FOR GENERATING THE ENTRY JUMP TABLE AT THE END OF THE MULTIENTRY
   UCOOE PROCEDURE AND EACH CALLED UCOOE PROCEOURE FOR EACH ENTRY POINT. *)
ENTRYREC = RECORD
              EXTPTR: POINTEXTNAME; (* POINTS TO ENTRY IN EXTERNAL TABLE ● )
              NUMARG,               (* # Of PARAMETERS FOR THIS ENTRY POINT *)
              ENTRYLABEL,           (* THE UCOOE LABEL THAT MARKS THE ENTRY
                                       POINT IN THE MULTIENTRY PROCEDURE *)
              ENTRYPOS: INTEGER;    (* THE POSITION IN THE MULTIENTRY
                                       SUBPROGRAM RELATIVE TO OTHER ENTRY
                                       STATEMENTS.  IF SUBPROGRAM HEADING,
                                       IT IS 0. *)
              HEAOENTADOR: POINTENTADOR; (* THE LIST OF PARAMETER ADDRESSES *)
              NEXT: ↑ENTRYREC;
             ENO;

(* THIS IS USED FOR FORMING A LIST THAT KEEP THE GLOBAL AOORESSES ASSIGNED
   FOR ENTRY PARAMETERS, IN THE ORDER OF THEIR APPEARANCES IN THE
   MULTIENTRY SUBPROGRAM HEADING OR ENTRY STATEMENT. THIS LIST IS POINTED
   TO FROM THE ENTRYREC RECORO THAT REPRESENTS EACH ENTRY POINT.  THE
   PURPOSE Of THIS LIST IS TO GENERATE THE COOE THAT COPIES AOORESSES OF
   ACTUAL PARAMETERS TO THEIR GLOBAL ASSIGNED LOCATIONS IN THE UCOOE
   PROCEDURE FOR EACH ENTRY POINT. ● )
ENTAOOR = RECORD
             ADDR: INTEGER;
             NEXT: ↑ENTAOOR;
           ENO;
```

The global pointer **HEADENTRYLIST** points to the list of the **ENTRYREC** records when processing a multiple entry subprogram. **HEADENTRYLIST** is reset to NIL at the start of each program unit.

18.4 Processing multiple entry subprograms

Procedure **ENTRYPROCESSING** processes an entry point definition. It is called from the **SUBROUTINE, FUNCTION** or **ENTRY** statement processors, the former two cases being the beginning of the multiple entry subprogram. Its job is to form an ENTRYREC node and fill in the information. The **ENTRYREC** node is then appended to the list pointed to by **HEADENTRYLIST.** ***No*** code is generated. The dummy argument list is then processed. The list of **ENTADDR** nodes formed is attached to the **ENTRYREC** node. If a dummy argument appears for the first time, a location in the global level is allocated for it.

Procedure **ENTRYSTMT** processes an **ENTRY** statement. Apart from calling **ENTRYPROCESSING,** it generates the U-Code label on site that marks the entry point represented by the ENTRY statement in the multientry procedure.

Since the jump table for the multientry procedure is at the end of the procedure, a **UJP** IS always issued as the first instruction of this procedure. This jump directs the branch to the code of the jump table.

The code related to the jump table is generated by procedure **GENENTJUMPS**, called from procedure **BLOCK** Preceeding the Jump table is the code to load the branch parameter and an **XJP** instruction which directs the jump with reference to the jump table. The Jump table is emitted by traversing the list of entries pointed to by **HEADENTRYLIST**.

The code for the entry procedures is generated by procedure **GENENTPROCS,** called at the end of procedure **BLOCK** One procedure is generated for each node in the list of entries. After the procedure heading, a series of **LOD-STR** is generated for the parameters to that procedure, for copying addresses to global locations. Then follows the code to call the multientry procedure with an integer parameter that conveys the entry point. For functions, the call will result in a value returned, and additional code to take in the value and in turn return it is emitted.

18.5 Example

The following example illustrates how a multiple entry function is compiled.

Fortran:

```
        FUNCTION SETVAL (P1) ENTRY
        SETVAL=P1
        RETURN
        ENTRY ASSIGN (P2)
        P2 = P1
        RETURN
        END
```

U-Code:

```
       SETVA077  ENT  R4  77  11     ;the multientry procedure
        LEX   1  1
        LEX   2 72
        LEX   3 73
        PSTR  A  R  77  0  36        ;receive the branch parameter
        UJP   L1002                  ;jump to branch code
       L1001 LAB  0                  ;label for normal entry point
        LOO   A  M  1  576  36       ;parameter P1 kept at level 1
        ILOD  R  0  36
        STR   R  M  77  0  36
        PLOD  R  M  77  0  36
        RET
       L1003  LAB  0                 ;label for the ENTRY statement
        LOO   A  M  1  612  36
```

```
 LOO  A  M  1 576 36
  IL00  R  0  36
  ISTR  R  0  36
 PLOO  R M  77  0  36
 RET
L1002  LAB  0                  ;label for the branch code
 LOO  J R  77  0  36           ;load branch parameter
 XJP  J L1004 L1005  0  1
L1004  CLAB  2                 ;jump table
 UJP    L1001
 UJP    L1003
L1005  LA8  0
 PLOO  R  M  77  0  36
 RET
 OEF  R  36
 OEF  M  72
 END  SETVA077
SETVA076  ENT  R  4  76  1 1   ;entry procedure for FUNCTION statement
 LEX    1   1
 LEX       2  72
 LEX       3  73
 PSTR  A  R  76  0  36         ;receive parameter P1
 LOO  A  R  76  0  36          ;load and store address for P1
 STR  A  M  1  576  36         ; in level 1
 MST      4                    ;call the multientry procedure
 LOC  J  36  0
 PAR  J  M   0  0  36
 CUP  R  77  SETVA077  1 1
 STR  R  M  76  0  36          ;receive value returned
 PLOO  R  M  76  0  36         ;return value received
 RET
 OEF  R  36
 OEF  M  72
 END  SETVA076
ASSIG078  ENT  R  4  78  1 1   ;entry procedure for ENTRY statement
 LEX    1   1
 LEX       2  72
 LEX       3  73
 PSTR  A  R  78  0  36         ;receive parameter P2
 LOO  A  R  78  0  36          ;load and store address for P2
 STR  A  M  1  612  36         ; in level 2
 MST      4                    ;call the multientry procedure
 LOC  J  36  1
 PAR  J  M  0  0  36
 CUP  R  77  SETVA077  1  1
 STR  R  M  78  0  36          ;receive value returned
 PLOO  R  M  78  0  36         ;return value received
 RET
 OEF  R  36
 OEF  M  72
 END  ASS16078
```

19. Subroutine and Function Calls

Calls to user-defined or standard or intrinsic functions occur in an expression, and calls to subroutines occur in a CALL statement. Procedure USERFUNC processes calls to user-defined functions or subroutines. Calls to standard or Intrinsic functions are processed by procedure STANDARD-FUNC. The ways in which these calls are processed are described below.

19.1 Processing parameters *in calls*

Dummy arguments of subroutines and functions are allocated addresses in their own stack frames. All parameters in Fortran are passed by reference. During execution of a subroutine or function, these addresses contain the addresses of the actual parameters. The actual storing of the addresses of the actual parameters into these locations during procedure invocations are done by the PSTR instructions at the beginning of a U-Code procedure. In U-Code, the addresses to be passed are put on the stack with the PAR instruction to indicate that they are parameters, and then the procedure is called.

The arguments in a call to a user-defined function or subroutine are processed in procedure PROCESS-ARGUMENTS. The way an address is passed to the called subprogram depends on the form of the actual parameter. For a simple variable, array name or an array element, its address is passed. For a constant, an expression or a string, a location in the global (level 1) memory IS allocated to store the final value, and the address of this location IS passed. For subprogram names, a double-word is allocated in the global memory in which the level and address of the U-Code procedure (generated using the LDP instruction) is stored, and the address of the double word is passed. For a dummy argument as parameter, which includes a subprogram name argument to be passed on, the address as stored in the parameter location IS passed.

7 9.2 Function call

Procedure USERFUNC is used to scan and process the arguments of a function or subroutine call and to generate the code that actually does the call.

This procedure counts the arguments with procedure COUNT-ARGUMENTS, generates an MST U-Code instruction that indicates the beginning and size of the stack for the call, processes the arguments with procedure PROCESS-ARGUMENTS, and generates the code for the call. The segment number for the CUP instruction is obtained from field SEGMENNUM of the symbol table for call to a statement function and from the field NUMBER of the external table for call to a subroutine or an external function. Procedure USERFUNC updates the external table when an external subprogram is called.

19.3 Subroutine call

Procedure CALL-STATEMENT scans and processes a subroutine call. It gets and Inserts the name of the subroutine into the symbol table. The data type for the subroutine is set to NONE explicitly after its insertion in the table, because otherwise FSYMBOL would insert the default Fortran type instead of NONE with the subroutine name. Procedure USERFUNC is then called.

19.4 Standard function calls

Standard function calls are implemented in three ways:

1. By a direct call to an equivalent U-Code standard function (CSP instruction).
2. By generating in-line code.
3. By a call to a function in the Fortran run-time package (CUP instruction).

A list of the functions and how they are implemented follows:

DESCRIPTION	NAME	ARGS	RESULT	U-CODE
absolute value	ABS	real	real	ABR
	IAB	int	int	ABI
	DABS	doubl	doubl	ABR
(mod)	CABS	complx	real	inline
truncation	AINT	real	real	CUP
	INT	real	int	CUP
	IDINT	doubl	int	CUP
mod	AMOD	real	real	inline
	MOD	int	int	MOD
	DMOD	doubl	doubl	inline
max	AMAXO	int	real	CUP
	AMAX1	real	real	CUP
	MAXO	int	int	CUP
	MAX1	real	int	CUP
	DMAX1	doubl	doubl	CUP
min	AMINO	int	real	CUP
	AMIN1	real	real	CUP
	MINO	int	int	CUP
	MIN1	real	int	CUP
	DMIN1	doubl	doubl	CUP
int to real	FLOAT	int	real	CVT
real to int	IFIX	real	int	CVT
transfer sign	SIGN	real	real	CUP
	ISIGN	int	int	CUP
	DSIGN	doubl	doubl	CUP
positive diff	DIM	real	real	CUP
(0 if a1<a2)	IDIM	int	int	CUP
doubl to real	SNGL	doubl	real	CVT
complex to real	REAL	complx	real	inline
complex imag to real	AIMAG	complx	real	inline
real to doubl	DBLE	real	double	CVT
real to complx	CMPLX	real	complx	inline
conjugate	CONJG	complx	complx	inline
exponential	EXP	real	real	CSP EXP
	DEXP	doubl	doubl	CSP EXP
	CEXP	complx	complx	CUP
natural log	ALOG	real	real	CSP LOG
	DLOG	doubl	doubl	CSP LOG
	CLOG	complx	complx	CUP
common log	ALOG10	real	real	CUP
	DLOG10	doubl	doubl	CUP
sin	SIN	real	real	CSP SIN
	DSIN	doubl	doubl	CSP SIN
	CSIN	complx	complx	CUP

cos		COS	real	real	CSP	cos
		DCOS	doubl	doubl	CSP	cos
		CCOS	complx	complx	CUP	
tanh		TANH	real	real	CUP	
	(IBM)	DTANH	doubl	doubl	CUP	
square	root	SQRT	real	real	CSP	SQT
		DSQRT	doubl	doubl	CSP	SQT
		CSQRT	complx	complx	CUP	
arctan		ATAN	real	real	CSP	TAN
		DTAN	doubl	doubl	CSP	TAN
arctan	(a1/a2)	ATAN2	real	real	CUP	
		DTAN2	doubl	doubl	CUP	

20. Statement Functions

Procedure STMT_FUNCTION scans and processes a statement function. The dummy arguments of a statement function are local to it. They have to be present in the symbol table when processing the function definition, and they must disappear after the declaration IS processed. if their names are the same as other variable names used in that program unit, they must be recovered in the symbol table. In order to do this, it is necessary to save the symbol table entries the dummy arguments replace. This is done by forming a list of records called DUMMY-LIST. The fields saved in these records are those in the symbol table that can possibly be altered while processing the statement function definition.

Procedure STMT-FUNCTION gets and inserts the name of the statement function in the symbol table with LEVEL field set to 5, and ADDRESS field set to 0. It processes the dummy arguments by calling procedure DUMMY-ARGUMENTS, which inserts them in the symbol table and records the old contents in the DUMMY-LIST records pointed to by HEAD-DUMMY. The dummy arguments are allocated addresses at level 5 in the same way as dummy arguments for program units are allocated at level 4.

A segment number is assigned to the statement function segment and code is generated for the head of the segment by calling procedure BLKCODE_GENERATION. Then procedure ASSIGNVALUE, which is also used in processing the assignment statement, is called to evaluate the expresston and store the result of the expression in the space reserved for the statement function name at level 5. In this process, temporary locations may be generated and, if so, they will be allocated in the level of the statement function.

Finally, code is generated for the return of the statement function, and the dummy arguments of the function are erased from the symbol table by calling procedure ERASE_DUMMYS, which also recovers the old contents in the symbol tabie from the DUMMY-LIST records.

21. DO Loop

A DO statement causes code to be generated at two different places at the positions corresponding to the DO statement and at the Fortran label that marks the end of the range of the DO loop. In the former, code is generated for the initialization of the index variable of the loop, as well as for the final value and step amount if they are expressions, and a U-Code label is emitted to mark the beginning of the loop. In the latter, code is generated to increment the index variable by the step amount, to check if it exceeds the final value, and to branch back to the label that initiates the loop if it does not exceed the final value.

A list of opened do-loops is built to control code generation for do-loops. This works in the form of a stack to keep track of the nesting of do-loops. Each time a new DO statement is processed, an entry is created for it in the stack. CURRENTDO is a global variable that points to the record of the most recently opened do-loop at the top end of the stack. There is a dummy entry that marks the bottom of the stack.

The end of the range of a do-loop is determined as follows. When a new label number IS defined, this is checked against the end label number of the innermost DO. If it matches, then the innermost DO is terminated, and the same check is continued for the next outer DO. This process terminates when the current label number is not the same as the label number of the DO in the top DO stack. The label is then checked against the end label numbers of all the remaining, outer DO's. If there is any match, indicating an illegal nesting of do-loops, an error is reported. Also, at the end of a program unit, if the bottom marker is not at the top of the DO stack, which indicates one or more do-loops have not been terminated, an error message is generated.

The DO stack is formed with the DOENTRY record of the form:

```
DOENTRY = RECORD
             PREVIOUS : ↑DOENTRY;         (* POINTS TO NOOE OF PREVIOUS
                                             NESTEO 00 *)
             CONTROLVAR : ↑SYMBOL;        (* POINTS TO SYMBOL TABLE ENTRY OF
                                             CONTROL VARIABLE *)
             STMTLABEL,                   (* FORTRAN LABEL THAT ENDS THE
                                             THE RANGE OF THE LOOP *)
             PCOOELABEL : INTEGER;        (* PCOOE LABEL INSERTED WHERE
                                             THE DO-LOOP BEGINS • )
             STEPKINO,
             UPPERKINO : OOKINO;
             EXPLEVEL,                    (* IF EXP. THE LEVEL OF THE TEMP
                                             LOCATION USED *)
             STEPAMOUNT,                  (* IF CONST KINO THEN CONST VALUE * )
             UPPERAMOUNT : INTEGER;       (* IF EXP KINO THEN TEMP LOC *)
             STEPMTYPE ,
             UPPERMTYPE : CHAR;           ( • MEM TYPE OF THE TEMP LOC USED * )
             STEPVAR, UPPERVAR : POINTSYMBOL; (* FOR VAR KINO *)
          END;
```

The DO loop routines are also used in processing implied loops in the READ and WRITE statements.

2 1.1 Do-loop initialization

Procedure DOSTATEMENT scans and processes a DO statement. It pushes an entry in the DO stack, gets the Fortran label that terminates the range of the do-loop and inserts it in the en try just created, processes the control part of the do-loop by calling procedure **DO-CONTROL** and generates a U-Code label indicating the beginning of the do-loop.

In procedure **DO_CONTROL,** the control variable IS located and inserted in the symbol table. Code IS generated for the computation of its initial value and storage in the variable's memory location. The values or addresses of the final and increment values are saved in the most recently created **DOENTRY** record. If either of these is an expression, then the address is that of a newly allocated temporary location. This type of temporary is never released, since jumping out of and back into do-loops is supported.

The initial value, step amount and final value can be arbitrary expressions, which will be coerced to integers. The evaluation of these expressions happens only once, before the loop is entered, so a change in any of the variables that make up the expressions will not affect the number of times the loop is iterated. If, however, the step or final value is a simple integer variable, then changing it will affect the number of times the loop is executed. The default value of the increment amount is 1 if none is specified.

2 t .2 Do-loop termination

Procedure **CLOSEDO** generates code for the termination of a do-loop. It is called by procedure **BLOCK** each time a Fortran label IS found in the source code and the stack of active DO's is not down to the bottom mark. It checks if the label just found corresponds to the Fortran label that terminates the range of a do-loop, stored in the top entry of the **DO** stack. If it does, code IS generated to increment the control variable and test for the termination of the loop. Once code for the current **DO** is generated, the previous entry in the stack becomes the new **CURRENT** and it is checked if the label in **LABNO** also indicates the end of its range. If it does, code is also generated for its termmation. This is repeated until the label in **LABNO** is not the end of the range of the current top **DO** record. Then, the rest of the **DO** records are checked, and any that should be terminated by **LABNO** causes an error message which indicates bad DO nesting.

This procedure also checks the kind of the statement that terminates the loop and gives an error if it is one of the following: **RETURN,** PAUSE, **STOP, DO, GOTO,** arithmetic **IF** and **ENTRY.**

The generation of code for the termination of the loop is done in procedure **GENCODE_FOR-DO.**

2 1.3 *Do-loop example*

Fortran:

```
          DO 10 I=3,(J+3),2
           . . .
           code
           . . .
    10    CONTINUE
     7    STOP
          END
```

U-Code:

```
          LDC  J 36 3
          STR  J M 1 5 0 4 3 6   ;store initial value of control variable
          LOO  J M 1 540 36
          LDC  J 36 3
          ADD  J                 ;evaluate loop termination value
          STR  J M 74  72  36    ;store termination value in a temporary location
         L1002 LAB 0             ;label to mark beginning of DO loop
```

```
 code  for statements inside loop
 ...
L1001  LA8 0          ;label to mark tnd of the range of the DO loop
 LOO   J M 1 504 36   ;load value of control variable
 INC   J 2            ;increment it
 STR   J M 1 504 36   ;update the control variable
 LOO   J M 1 504 36   ;load control variable back
 LOO   J M 74 72 36   ;load loop termination value
 GRT   J              ;compare them
 FJP   L1002          ;jump back if still smaller
```

22. GOTO Statcmcnts and Statement Labels

FORMAT statement labels are entered both in the label table and the symbol table. All other labels are inserted only in the label table. The first time a label occurs, a U-Code label is assigned to it and inserted in the label table.

The check as to whether a statement label referenced is defined or not can be made only at the end of a program unit, since the left- and right-hand-side occurrences are processed independently. Procedure LABEL_LHS_CHECK IS called at the end of every program unit to search through the label table. For each label used only on the RHS but not on the LHS, a warning is given and the U-Code label is generated at the end of the code for the program unit with traps. Jumps to the undefined statement labels during execution will then cause a halt.

The three kinds of GOTO statements are processed as follows:

22.1 Unconditional GOTO

A simple UJP instruction is made to the corresponding U-Code label.

22.2 *Computed GOTO*

This compiles into the XJP instruction, which corresponds to the CASE statement of Pascal. First, code to load the branch variable is generated by calling procedure LOAD_VAR, which takes care of cases that the variable is simple, dummy or is an array element. The XJP instruction is then generated, with the branch table immediately following. This contains a list of UJP's for the statement labels.

22.2.1 Example

```
Fortran:   GOT0 (10,20,30),I

U-Code:
           LOO J M 1 SO4 36          ;load variable I
           XJP J L1001 L1002 1 3     ;jump according to table
          L1001 CLAB 3               ;jump table of length 3
           UJP  L1003                ;jump to statement 10
           UJP  L1004                ;jump to statement 20
           UJP  L1005                ;jump to statement 30
          L1002 LA8 0                ;end of jump table

           call to execution error routines
```

22.3 Assigned GOT0

Because U-Code labels referenced in U-Code jump instructions must be label names, code for this Fortran statement IS somewhat Inefficient.

There are two ways this statement could be compiled into U-Code. The first IS to use the XJP instruction, which IS like transforming the assigned GOTO statement into the corresponding computed GOTO. The second method, which is the one used, does not use XJP, and generates denser U-Code. The label variable is multiply loaded and its value compared one by one with each statement label in the list until equality is found. Then the corresponding jump is made.

If the label variable is a simple variable, the multiple loading is done by calls of LOADVAR. If it is an array element, the subscript expression must be evaluated only once. Thus, LOADVAR is called only once, and the value loaded is saved in a temporary location. The value stored in this location is then multiply loaded.

22.3.1 Example

```
Fortran:    GOTO J,(10,20,30)

U-Code:
            LOD   J M 1 504 36        ;load label variable J
            LDC   J 36 10             ;load constant 10
            NEQ   J                   ;compare
            FJP   L1001               ;if equal, jump to statement 10
            LOD   J M 1 504 36        ;load J back
            LDC   J 36 20             ;load constant 20
            NEQ   J                   ;compare
            FJP   L1002               ;if equal, jump to statement 20
            LOD   J M 1 504 36        ;load J back
            LDC   J 36 30             ;load constant 30
            NEQ   J                   ;compare
            FJP   L1003               ;if equal, jump to statement 30
```

23. The Arithmetic IF and Logical IF Statements

23.1 *Logical IF*

The logical IF is the only type of Fortran statement that is *compound.* The compilation is separated into two parts. The first part (procedure LOGICALIF) processes the logical expression enclosed by the parentheses. Procedure LOGICALEXPR is called which will generate the U-Code that evaluates the IF condition and puts the result on top of the stack. The outermost pair of parentheses is not checked here since they have been checked inside procedure CLASSIFY. The global variabie IFDEST serves as a flag to indicate whether current processing is inside a logical IF statement. it is initialized to -1 in procedure INITBLOCK. When a logical IF statement is encountered, it is set to the number of the U-Code label which will be generated a& the end of the whole IF statement. Code is generated to jump to this label if the IF condition is false.

The second part is compiled as an independent Fortran statement, the only difference being that IFDEST is set, and consequently a new statement is not read in from the source file. A check is made if the type of the statement is among those allowed as the second part of a logical IF statement. After the second part of the logical IF IS complied, the U-Code iabei IFDEST is generated and IFDEST is reset to -1.

Note that because the second part is processed as an independent statement, other statement processing procedures cannot assume that the iexemes for the statement start at position 1.

23.2 Arithmetic IF

The arithmetic expression in the first part of this IF statement is processed by calling procedure ARITH, which will generate the U-Code to evaluate the arithmetic expression and put the result on top of the stack. Again, the outer pair of parentheses is not checked since they are checked inside CLASSIFY.

Note that because of the *three-way branch,* two tests have to be made of the value on top of the stack. Since the value disappears after a test, code is first generated to store the top-of-stack value in a temporary location. Then follows code to make the tests and do the jumps. The form of the U-Code generated is:

```
Fortran:   IF (J+3) 10,20,30

U-Coda:
           LOO   J M 1 504 36        ;load varablr J
           LDC   3 36 3              ;load constant 3
           ADO   J                   ;compute (J+3)
           NSTR  J M 74 72 36        ;save result
           LDC   J 36 0
           GEQ   J                   ;compare result with 0
           FJP   L1001               ;jump to statement 10 if < 0
           LDC   J 36 0
           LOD   J M 74 72 36        ;load result back
           NEQ   J                   ;compare result with 0 again
           FJP   L1002               ;jump to statement 20 if = 0
           UJP   L1003               ;otherwise, jump to statement 30
```

24. The PRINT Statement

The syntax of this statement is based on the way the Pascal standard output routines are called, so the processing of this statement is done in a straightforward manner.

After generating the call co the Pascal output initialization routine, a loop is entered which iterates for the list of output Items. in each iteration, ARITH is first called to leave the expression on top of the stack. The following lexemes are scanned co check for any specification of output field width. The corresponding Pascal output routine is then called according to the type of the expression evaluated: string, integer, boolean, reai or complex.

24.1 *Example*

```
Fortran:    PRINT 'X=',5:1,'C=',(3.,2.)

U-Code:
            LDA    M 1 117 9   ;load address of file OUTPUT
            CSP   A SIO    11
            LCA    M 1 8 'X='
            LDC   J 36 2
            LOC   J 36 2
            CSP   A WRS     4 1   ;write 'X='
            LOC   J 36 5
            LDC   J 36 1
            CSP   A WRI     3 1   ;write integer 5
            LCA    M 18 'C='
            LDC   J 36 2
            LOC   J 36 2
            CSP   A WRS     4 1   ;write 'C='
            LOC   R 36 3.0
            LDC   R 36 2.0
            STR   R M 74 72  36   ;store imaginary part temporarily
            LOC   J 36 14
            LOC   J 36 9
            CSP   A WRR     4 1   ;write real part
            LOO   R M 74 72  36   ;load back lmagrnary part
            LDC   J 36 14
            LOC   J 36 9
            CSP   A WRR     4 1   ;write imaginary part
            CSP   A WLN     1 1
            CSP   PEIO      1 0
```

2 5. FORMAT Statement Processing

FORMAT statements are processed in two stages. First, the FORMAT statement is scanned and the information for the FORMAT statement IS entered in a created FORMTLIST record. The list of these records about the FORMAT statements in the various program units is pointed to by the global variable HEADFORMTLST. The structure of the FORMTLIST record is:

```
FORMTLIST = RECORO
                PTRFMTSTR:↑FORMTSTR;(* POINTER TO THE FORMAT STRING LIST *)
                NEXT:↑FORMTLIST;
                AOORESS,
                LEVEL:INTEGER;(* ADDRESS WHERE FORMAT STRING IS STORED ●    )
              END;
```

The FORMAT string specification is also saved in a list formed with records called FORMTSTR with the structure:

```
FORMATSTRING . PACKED ARRAY [1..MAXCHARINLCA] OF CHAR;

FORMTSTR = RECORD
               STR:FORMATSTRING;              (* FORMAT STRING *)
               NEXT:↑FORMTSTR;
             END;
```

The purpose of this second list is to save space. Only increments of MAXCHARINLCA units of storage need be allocated by the complier. The constant MAXCHARINLCA defines the limit on the length of the literal allowed in the U-Code LCA Instruction. Currently, it is 64. Thus, another advantage of this scheme is that the characters on each record can be loaded by a single LCA instruction.

25.1 The FORMAT statement

Procedure FORMAT-STMT scans and processes a FORMAT statement. It gets the label of the FORMAT statement in character form and inserts it into the symbol table indicating that it IS a FORMAT label. An address is allocated to the FORMAT label which holds the address of the location where the FORMAT string specification is stored.

A new entry in the list of formats, FORMTLIST, is created and the following information IS obtained and inserted: (a) the address and level assigned to the FORMAT label and (b) the pointer to the FORMAT string specification list.

The FORMAT string specification is copied into the FORMSTR list character by character. Any unused space in the last FORMTSTR record is cleared to blanks.

25.2 Initialization of formats

Procedure INIT_FORMATS is used to generate code for &he loading of the FORMAT string specifications into memory at execution time. This procedure is called by procedure VARINITIALIZATION which is in charge of the initialization of variables for the compiler. (See Section 9.5.)

For each FORMTLIST record, procedure INIT_FORMATS generates a series of LCA-LDA-MOV instructions according to the length of the FORMTSTR list. By the sequences of the three instructions, the segments of each FORMAT string stored in the FORMTSTR records are moved to be adjacent to each other in a block starting at address DISPLACEMENT, level 3. The LDA-STR instructions then follow which stores the address where the FORMAT string begins at the address of the FORMAT label.

26. Read and Write Statements

26.1 Run-time I/O routines

Fortran allows lists and loops within the READ and WRITE statements. In order to manage the fairly complex variable sequences, the implementation uses multiple calls to system routines listed below:

26.1 .1 Initialization of I/O routines

The run-time routines require initialization at the start of execution of any Fortran program. Therefore, a call to

```
FILEI031
```

is always generated at the beginning of a Fortran program. This initializes the file table which describes the status of each file or device. All of them are assumed to be closed. The file for the output of execution *error* messages is opened. An error flag for the I/O run-time routines is initialized.

26.1.2 Initiaiization of single I/O statement

One call to an initialization routine before executing each READ/WRITE statement is required before any data transmission call can be made.

```
READI028
WRITI025
```

Parameters: integer device number and address of FORMAT string.

The device (or file, as the case may be) is opened if not already opened in the corresponding mode. In output, the cursor to the I/O buffer is initialized. In input, the first line is read into the I/O buffer. If the FORMAT pointer is not NIL (unformatted I/O), the variables for processing the FORMAT string are initialized.

26.1.3 Data transmission

Each call transmits one value, using one entry from the FORMAT description. These calls may be embedded in lwps within the calling program, such loops being invisible to the I/O routines.

```
READV030
WRITV027
```

Parameters: address of data value,
size of data value in bytes and
coded type of data value (0 integer,1 real,2 logical).

These routines scan the FORMAT string untii the next I/O field is found, and service the

FORMAT string's contents as it scans past them. The value is transmitted according to the field description (which also implies the type of the data value), taking into account the size of the variable given as the 2nd parameter. If I/O is unformatted, then the 3rd parameter (type) is taken into account to determine the desired conversion.

26.1.4 Termination

These calls finish the transmission for each READ/WRITE statement, release buffers and return an error code. Any further I/O has to begin with initialization calls.

READT029
WRITT026

Parameter: address of indicator.

The FORMAT string is scanned until the end or the next I/O field if it occurs first. In output, the I/O buffer is written out. The indicator is a quarter-word in the global memory and is set to one of the following:

0. I/O perceived correct
1. I/O error detected
2. I/O end of file detected

26.1.5 Rewind

Lastly, a call to

REWIN032

Parameter: file number

is generated at a REWIND or OPEN statement in the Fortran source program: This causes a reset if the file has been reset before, or a rewrite if the file has been rewritten before. Otherwise, no operation is performed. This enables the user to start at the beginning of the file again for the same operation on the file.

26.2 Compiler routines

Procedure IO-STATEMENT scans and processes the READ/WRITE statements. Parameter READING to this procedure indicates the kind of I/O statement, being TRUE for a READ statement and FALSE for a WRITE statement.

The general form for the I/O statements is :

```
READ (DEVICE,FORMAT) list

READ (DEVICE) list            ; if unformatted
```

where list is a list of variables that may only include simple variable names, array names and array elements. OEVICE is the device number and FORMAT may be a FORMAT statement label or an array name.

For the I/O of arrays, when no control variable is explicitly established, two temporary locations are obtained. These temporary locations, pointed to by variables MAXPRINTARRAY and **CONPRINTARRAY,** store the upper bound (number of elements in the array) and index respectively for the array.

Procedure IO_STATEMENT gets &he device number and the **FORMAT** specification *(either* **a** **FORMAT** statement label or an array name), and generates the code to call the run-time routines for the initialization of the I/O of the current statement, the code for data transmission of the variables {by calling procedure **LIST-PROCESSING**) and the code to call the routines for the termination of the I/O for the statement.

Procedure **LIST-PROCESSING** processes the variables in an I/O statement. It is called by procedure IO_STATEMENT the first time, and by itself recursively when an implied DO or another list of variables surrounded by parentheses is found in the list being processed. Parameter **IN-DO-IMPLIED** indicates if the list of variables being processed belongs to an implied DO **or** is just a list of variables surrounded by parentheses.

LIST-PROCESSING tooks at each element of the list. if it is a simple variable, array element or an array, procedure **VARNAME** is called. If it is **an** implied DO, which is detected by procedure **CHECK-DO-IMPLIED,** procedure **DO_IMPLIED** IS called to process it. If it is **a** simple list, procedure **LIST-PROCESSING** IS called recursively to process this inner list, with **IN-DO-IMPLIED** set to false.

Procedure VARNAME generates the code for the I/O of a simple variable, array element or a complete array. For the simple variable or array element, the parameters to the system routine that does the data transmission are loaded and then **a call** to it is generated. For the complete array, a special loop in U-Code is generated. This loop is preceded by, in their order, the code to compute the number of elements of the array and store it in MAXPRINTARRAY, the code to initialize CONPRINTARRAY, the indexing location, to 0 and a U-Code label to mark the beginning of the loop. Inside the loop is the code to load the parameters for the system routine and a call to it. The address of each element of the array is computed by loading the initial address of the array and then indexing it with the value at **CONPRINTARRAY. At** the end of the loop is the code which increments the index and tests its value against that in MAXPRINTARRAY for loop termination condition.

Procedure DO-IMPLIED processes an implied DO. First, it processes the control part of the do-loop using procedure **DO_CONTROL;** then it generates the code for the list of variables in the Implied DO by calling procedure **LIST_PROCESSING** with the parameter **IN-DO-IMPLIED ser to** true; after this it generates the code to close the do-loop using procedure CLOSEDO. Each implied DO has associated a dummy Fortran label {above 100000 to avoid any possible duplication with an existing Fortran label) that is used by the CLOSED0 routine. These dummy labels are not Inserted into the label table.

26.3 *Code* generated

```
Fortran: INTEGER C(3,3),P(5)
         READ (4,8) (C,(P(I),I=N,M,1))

U-Code:
         MST      4                       ;initialiation:
         LDC   J  36  4                   ;load device number
         PAR   J  tl  0  0  36
         LOO   A  M  1  1008  36          ;load address of FORMAT string
         PAR   A  M  1  1008  36
         CUP   P  28  READ1028  2  0      ;call initialization routine

         LDC J  36  9                     ;I/O of array c
```

```
      STR  J M 74 72 36         ;store sire of af array C in MAXPRINTARRAY
      LDC  J 36 0
      STR  J M 74 108 36        ;load initial value in COMPRINTARRAY
I.1001 LAB 0                    ;label for beginning of generated loop
      MST   4
      LDA   M 1 504 0           ;load address of array element:
      LOD  J M 74 108 36
      IXA  J 36
      PAR  A M 0 0 36
      LDC  J 36 4               ;load size of data value
      PAR  J M 0 0 36
      LDC  3 36 0               ;load coded type
      PAR  J M 0 0 36
      CUP  P 30 READV030 3 0    ;call data transmission routine
      LOO  J M 74 108 36        ;load control variable from CONPRINTARRAY
      INC  J 1                  ;increment it
      STR  J M 74 108 36        ;update it
      LOO  J M 74 108 36        ;load it back
      LOO  J M 74 72 36         ;load final value from MAXPRINTARRAY
      GEQ  J                    ;compare
      FJP   L1001               ;jump back if smaller

      LOO  J M 1 1152 36        ;implied 00 loop:
      STR  J M 1 1116 36        ;save initial value in control variable
L1002 LAB 0
      MST   4
      LDA   M 1 828 180         ;load address of current element of P
      LOO  J M 1 1116 36
      OEC  J 1
      IXA  J 36
      PAR  A M 0 0 36
      LOC  J 36 4               ;load size of data value
      PAR  J M 0 0 36
      LOC  J 36 0               ;load coded type
      PAR  J M 0 0 36
      CUP  P 30 REAOV030 3 0    ;call data transmission routine
      LOO  J M 1 1116 36        ;load control variable
      INC  J 1                  ;increment control variable
      STR  J M 1 1116 36        ;update control variable
      LOO  J M 1 1116 36        ;load back control variable
      LOO  J M 1 1188 36        ;load termination value
      GRT  J                    ;compare
      FJP   L1002               ;jump back if not reached

      MST   4                   ;I/O termination:
      LDA   M 1 396 36          ;load address of indicator
      PAR  A M 0 0 36
      CUP  P 29 REAOT029 1 0    ;call to I/O termination routine

       code to check value of indicator returned and
       trap execution If in error
```

27. The Fortran I/O Run-time Package

The Fortran I/O run-time routines are used for the execution of READ and WRITE statements. These routines are written in Pascal and make use of the lowest level Pascal I/O run-time routines.

The I/O routines require the double precision facility in Pascal to properly process the I/O of doubie precision variables in Fortran. When this facility is not available, double precision I/O may be processed only up to the accuracy allowed by single precision. The I/O requirements of quarter- and half-word variables are completely handled.

The I/O routines are stored in loader format along with the intrinsic and standard function run-time routines, and linked to the main program by the linker for execution.

27.7 Structure of the I/O package

The separate parts that make up the I/O run-time package are listed with their procedures in the order as they appear in the program:

1. error procedure - This outputs I/O execution error messages and sets error flags:
   (a) procedure ERROR.

2. routines to handle the operations of the I/O buffer:
   (a) procedure CALLNEWOUTLINE;
   (b) procedure NEWOUTLINE;
   These write out the buffer as the next line in the output file.
   (c) procedure CALLNEWINLINE;
   (d) procedure NEWINLINE;
   These input the next line in the input file into the buffer.
   (e) procedure PUTCHAR - This puts the next output character to the I/O buffer;
   (f) procedure GETCHAR - This gets the next input character in the I/O buffer.

3. procedures to process the FORMAT string:
   (a) procedure NEXTFIELD - When called, it will scan the format string starting from where it was before, processing what it encounters until it gets to the next I/O field. The specifications of the field are returned.

4. procedures for output conversions of data values:
   (a) procedure PRIFIELD - prints an integer in an I-formatted field;
   (b) procedure PRFFIELD - prints a real number in an F-formatted field;
   (c) procedure PREFIELD - prints a real number in an E-formatted field;
   (d) procedure PRGFIELD - prints a real number in an G-formatted field;
   (e) procedure PRLFIELD - prints a boolean in an L-formatted field;
   (f) procedure PRAFIELD - prints the contents of a variable in an A-formatted field.

5. procedures for formatted input conversions of data values:
   (a) procedure REIFIELD - reads in an integer in an I-formatted field;
   (b) procedure REEFGFIELD - reads in a real number in an E-, F- or G-formatted field, the effect being defined as identical;
   (c) procedure RELFIELD - reads in a boolean from an L-formatted field;
   (d) procedure REAFIELD - reads in the characters in an A-formatted field to a variable.

6. procedures for unformatted input conversions of data values:
   (a) procedure UNFINTINPUT - scans and inputs an integer;
   (b) procedure UNFREALINPUT - scans and inputs a real number;
   (c) procedure UNFBOOLINPUT - scans and inputs a booiean.

7. procedures called externally:
   (a) procedure WRITINI (U-Code name is READI 026);
   (b) procedure WRITTRM (WRITI023);
   (c) procedure WRITVAL (WRITV025);
   (d) procedure READINI (READI026);
   (e) procedure READTRM (READT027);
   (f) procedure READVAL (READV028);
   (g) procedure FILEINI (FILEI029);
   (h) procedure REWIND (REWIN030).

In WRITVAL and READVAL, for formatted I/O, 3 NEXTFIELD is first called, followed by the appropriate procedure in 4 or 5. For unformatted I/O, in WRITVAL, the standard field width is assigned and the appropriate procedure in 4 (a), (c) and (e) is called. In READVAL, the appropriate procedure in 6 is called.

Note that the procedures in 4, 5 or 6 treat the transmitted data value in double-word size. WRITVAL will do the necessary shifting for data values of smaller sizes before calling 4 READVAL will do the necessary shifting after calling 5 or 6. PRAFIELD and REAFIELD, however, are exceptions since the number of transmitted characters is different for variables of different sizes (four characters per single-word, 9 bits for each character). These two procedures are called from WRITVAL and READVAL with an extra parameter that gives the size information of the variable.

27.2 *Processing the FORMAT string*

The entities allowed in a FORMAT string are: numbers, Hoilerith string, literal string (enclosed in quotes), comma, slash, X, the left and right parentheses, P, and the field specifications for I, E, F, G, L, A fields. Items enclosed in parentheses form a *group*. The number of groups in the same level is not limited, but only three levels of grouping are allowed, including the outermost group which is the FORMAT string itself.

Procedure NEXTFIELD is in the form of a loop which scans and processes one of the above entities each round. Two booleans COMMAED and COUNTED keep track of the syntactic information in checking for syntax errors. The comma is not mandatory in the FORMAT string in cases where its absence causes no ambiguity.

Variables GPCOUNT2 and GPCOUNT3 keep track of the current position of the cursor within groups. When GPCOUNT3 is 0, the cursor is not within a 3rd level group. When the cursor is within a 3rd level group, GPCOUNT3 indicates the number of times it still has to scan across that group. It is decremented each time the end of the 3rd level group is reached. The same holds for GPCOUNT2 and 2nd level group. GPBEGIN1, GPBEGIN2 and GPBEGIN3 give the starting position of the current groups of the respective levels.

When the scanning has reached the end of the FORMAT string but still has yet to look for the next I/O fieid, back-up is made to the beginning of the last 2nd level group. For this purpose, LASTGPPOS and LASTGPREP will hold the starting position of the last 2nd level group (or the 1st level group - the FORMAT string itself, if no 2nd level group exists) and its repetition factor.

To prevent NEXTFIELD from looking for a field indefinitely when in fact no field exists from its back-up point to the end of the FORMAT string, the boolean variable FIELDFOUND is used. Whenever the end of the FORMAT string is reached, there will be back-up only if FIELDFOUND is true. FIELDFOUND is set false when scanning the beginning of the FORMAT string and at the beginning of every 2nd-level group that can possibly be the back-up position for the FORMAT string. It is set to true whenever a field is found.

At the end of the I/O statement (when procedure WRITTRM or READTRM is called), NEXTFIELD has to be called the last time to bring the scan to the next I/O field or the end of the FORMAT string. Here, FIELDFOUND is first set to be false before calling NEXTFIELD so that no backing up is done at the end of the FORMAT string.

2 7.3 I/O management

An I/O buffer of fixed length (currently 256 characters) is maintained. This stores the next output line being built, or the next input line from the input file. In output, the buffer is written to the output file when a new output line is specified. In input, the next line from the input file is read to the buffer when the next input line is specified.

The length of the output or input line is variable. If the output line exceeds the length of the I/O buffer, a next output line is automatically created to accomodate the extra characters. if the input line exceeds the length of the I/O buffer, the input line still assumes its length, but the characters to the right of the line limit that cannot be accomodated within the buffer are ail taken to be the blank character.

27.4 Internal-external correspondence of data values

In standard Fortran, the type of conversion in formatted I/O is determined by the field type in the FORMAT s&ring, and not according to the type of the variable in the READ or WRITE statement. The same content (bit pattern) of the location in I/O is to be treated as different types of data values according to the field types specified. (This is necessary since, for instance, no string variable exists but the character type field (A-field) does exist.) The Fortran user has to make sure that his variables in formatted I/O have the right corresponding field type in the FORMAT string for the correct values to be transmitted.

In the implementation, the data type

```
IOCOC = RECORD
           CASE INTEGER OF
           0: (INTVAL: INTEGER);
           1: (REALVAL: REAL);
           2: (CHARVAL: ARRAY[1..4] OF CHAR);
           3: (BOOLVAL: BOOLEAN)
        END;
```

allows the decoding of the content of a memory location as different types of data values. The above default is implemented by making a variable of this type as the reference parameter for the I/O variable in the externally called procedures READVAL and WRITVAL. After calling NEXTFIELD, the type of conversion is known from the field type, and the corresponding conversion procedure is called using the suitable variant field as the parameter.

The size of the variable (one of the parameters in READVAL and WRITVAL) is taken account of by shifting the value prior to output conversion or after input conversion. In formatted I/O, the form of the input or output field has no correspondence to the variable size. In output, E-field and D-field differ only with respect to whether E or D indicates the exponent. In input, D or E makes no difference in indicating the exponent.

27.5 Output conversions of data values

Ail output conversions can be treated as formatted, unformatted output being simply formatted output with standard field sizes for the different types. The standard field sizes are those that allow the full content of the variable location to be displayed. Thus, they vary with the size of the variable.

In ail output conversions, variable IOBUFCURS always points to the left boundary of the output field. Another variable WI indexes across the width of rhe field. The FOR loop is always used, and WI is the control variable.

Here are details for the output conversion of real numbers:

The real number is first normalized to >= 0.1 and < 1.0, the power being accumulated in the integer variable E. Rounding is performed at the appropriate place by adding 0.5 to the appropriate power of ten to the digit after the least significant printed digit. Truncation then does the desired rounding.

For conversion to character form, the normalized mantissa is multiplied by 10 ** 11 (given MAXINT = 34359738367 has I I digits) if < .34359738367, and by 10 ** 10 otherwise, to convert to an in teger. This arrangement is made to preserve as much accuracy as possible. The output characters are then made from this integer. This integer only gives the significant digits. The position of the decimal point is monitored by E, taking into account the exponent to be printed. Thus, even if the output mantissa has more than ii digits before the decimal, the less significant digits are made ail zero.

The algorithm for output conversion of E-field (similar for F-field with slight modifications) is: (W, D and S are the field descriptors)

```
1. IF (0 > (W-O-5) ) OR (OUTREAL < 0) AND (0 < (W-O-6) ) OR
      (S > (W-O-5)) OR (OUTREAL < 0) AND (S < (W-O-6))
   THEN print '*' across field
   (field not large enough)

2. ELSE IF (OUTREAL < MINREAL) AND (OUTREAL > -MINREAL)
   THEN print zero
   (MINREAL is the smallest magnitude of real number alioued.
    Note that this is different from the smallest representable
    rea l number, which has the louest power but uithout the
    mantissa normalized.)

3. ELSE (a) get sign if negative
        (b) normalize OUTREAL to >= 8.1 and < 1.0, and
            accumulate the power in variable E
        (c) I F ((S+D)>= 0) AND ((S+D) <= 10)   (Here, 10 is
                largest number of significant digits stored in
                a uord of memory)
            THEN OUTREAL := OUTREAL + 0.5 * 10 ** (-(S+D))
            (Do round i ng. (S+D) is the number of significant
     digits printed.)
(d) IF OUTREAL > 1.0 (increase to > 1.0 due to rounding)
    THEN BEGIN OUTREAL : = OUTREAL / 10;
                E : = E + 1 END
(e) IF  OUTREAL < .34359738367
    THEN CURTRUNC:= TRUNC(OUTREAL * (10 ** 11))
    ELSE CURTRUNC := TRUNC (OUTREAL * (10 ** 10))
(f) output digits from CURTRUNC, the decimal point being
    governed by S.
(g) E := E - S;
    print the exponent according to E.
```

27.6 Input conversion of data values

In unformatted input conversion, the input file is scanned line by line until the next non-blank character is found, and decoding starts from this position. Blanks and end-of-line separate input entities.

In formatted input conversion, variable IOBUFCURS always points to the left boundary of the input field. Variable W1 indexes across the width of the field. For integer and real inputs, blanks in a field imply 0. For real input, presence of '.' overrides the implicit decimal place indicated by D in the field specification. Presence of the exponent overrides the effect of the scale factor S. Effects of D-, E-, F- and C-formatted fields are defined as identical in real input.

The loop that processes the input characters (with one character look-ahead) is always of the form:

```
WHILE (BUFFER[W1] IN [set of looked-for char]) AND
      (W1 is within boundary) DO
   BEGIN
      process this character
      W1 := W1 + 1
   END;
```

where boundary refers to the field boundary (or the decimal boundary within the field) in formatted input and line boundary in unformatted input.

This arrangement requires that the input buffer be declared one unit longer to prevent out-of-bounds error of the buffer index. Another possible arrangement (not used) which does not entail this extra declaration requires an extra flag and less straightforward structure:

```
DONE := FALSE;
WHILE NOT DONE DO
   IF BUFFER[W1] IN [set of looked-for char]
      THEN BEGIN
              process this character
              W1 := W1 + 1;
              IF W1 not within boundary
                 THEN DONE := TRUE;
           END
      ELSE DONE := TRUE:
```

Input digits are always decoded into an integer variable, even if the digits belong to the mantissa of a real number.

To check for overflow error and to ensure that any representable integer can be input, the scheme used is: (Given MAXINT = 34359738367)

```
KEEPNUM : = 0;
WHILE (NXTCHAR in ['0'..'9']) DO
   BEGIN
      IF (KEEPNUM > 3435973836) OR
         ((ININT = 3435973836) AND (NXTCHAR IN ['8', '9']))
         THEN over f l ow-error
         ELSE KEEPNUM := KEEPNUM * 10 + (ORD(NXTCHAR) - ORD('0'));
      get NXTCHAR
   END:
```

In reading real numbers, the input is decoded into the integer variable KEEPNUM which keeps the mantissa and integer variable E which keeps the exponent such that KEEPNUM ** E gives the correct real value. In this case, too many digits in the mantissa should not cause overflow if still representable as a real number. Here, the decoding part of the WHILE loop that processes the digits in the mantissa is:

```
IF (KEEPNUM > 3435973836) OR
   ((KEEPNUM = 3435973836) AND (NXTCHAR IN ['8', '9']))
   THEN E := E + 1
   ELSE KEEPNUM := KEEPNUM * 10 + (ORD(NXTCHAR) - ORD('0'));
```

(If current digit is after the decimal, then increment of E above is not necessary.)

In practice, the IF condition above can be replaced by just IF (KEEPNUM >= 3435973836) for greater efficiency without much loss of accuracy.

References

[AKe80 J **J. Allchin and A. Keller:** *FLASH: A Language-independent, Portable File System, S-l* project document, Jan 1980.

[ANS64] **American Standard Association, X3.4.3:** *Fortran vs. Basic Fortran,* Comm. of the ACM, Vol. 7, No. 10, October 1964, pp. 591-625.

[ANS66] **ANSII:** *USA Standard Fortran,* USA Standards Institute, USAS X3.9-1966, New York 1966.

[ANS71 J **Americal National Standards Committee X3J3:** *Clarification of Fortran standards - second* report, Comm. of the ACM, Vol. 14, No. 10, October 1971, pp. 628-642.

[ANS76] **American National Standards Committee X3J3:** *Draft Proposed ANS Fortran,* Sigplan Notices, Voi. 11, No. 3, March 1976,(254 pages).

[BrW68] **Gary Y. Breitbard and Gio Wlederhoid:** *PL/ACME: An incremental Compiler for a Subset of PL/1,* information Processing 1968 (Proceedings of the 1968 IFIPS Conference, Edinburgh), North Holland, 1969, pages 358-363.

[Bsh79] **Randy Bush:** *UASMINT: A U-Code Assembler and Interpreter,* S-l project document, June 1979.

[CCN79] **F. Castaneda, F. Chow, P. Nye, D. Sleator and G. Wiederhold:** *PCFORT - A Fortran to P-Code Translator,* CSL Technical Report 160, Stanford University, Jan 1979.

[FiZ78] **Jim Finnel and Polle T. Zellweger:** *The S- 1 Multi-processor,* OSL Technical Note 142, Stanford University, June 1978.

[GiW77] **Erik J. Gilbert and David W. Wail:** *P-Code Intermediate Assembly Language.* S-l project document PAIL-3, 18JUL77.

[GNR79] **Phillip Geering, Peter Nye, Armando Rodriguez and** *Arthur Samuel: S- 1 U-Code: A Universal* P-Code *for* the S- 1 *Project,* S-l project document PAIL-6, August 1979.

[GWa78] **Erik J. Gilbert and David W. Wall:** *Specification for Run-time Support for Pascal,* S-l project document PRUN-0, 20MAR78.

[Gri71J **David Gries:** *Compiler Construction for Digital Computers.* John Wiley and Sons, 197 1, pp. 304-3 12.

[HiN80] **Bruce Hitson and Peter Nye:** *Run-time Specification for a Pascal U-Code System,* S-l project document PRUN-1, Dec 1979.

[JeW75] **K. Jensen, and N. Wirth:** *Pascal User Manual and Report,* Springer Verlag, New York, 1975.

[KeW79] **Arthur Keller and Gio Wiederhold:** *S-i intermediate Loader Format and S- 1 Linker,* S-l project document LDI-8 & SLIM-O, Dec 1979.

[NAJ75] **K. Nori, U. Amman, K. Jensen, et al.:** *Pascal P Compiler Implementation Notes,* ETH Zurich, 1975.

[Org66] **Elliott I. Organick:** *A Fortran IV Primer,* Addison-Wesiey, 1966, p.48.

[PSi79] **Daniel R. Perkins and Richard L. Sites:** *Univeral P-Code Definition,* version [0.3], UCSD/CS-78/037, July 1979.

[SPT79] **Richard L. Sites, Daniel R. Perkins, J. Richard Tinling and John B. Collings:** *Machine-independent Pascal Optimizer Project: Final Report,* UCSD/CS-79/038, Nov 1979.

[WiB70] **Gio Wiederhold and Gary Breitbard:** *A Method for increasing The Modularity of Large System,* IEEE Computer, Vol. 3, no. 2, March-April 1970, page 30.

[Zel80] **Polle Zellweger:** *S-1 Code Generator and Optimizer,* S-l project document SOPADOPE-2, Jan 80.