COMPUTER SYSTEMS LABORATORY

STANFORD ELECTRONICS LABORATORIES
DEPARTMENT OF ELECTRICAL ENGINEERING
STANFORD UNIVERSITY. STANFORD. CA 94305

STAN-CS-79-7 14

PCFORT

A Fortran-to-Pcode Translator

Fernando Castaneda
Frederick Chow
Peter Nye
Dan Sleator
Gio Wiederhold

TECHNICAL REPORT NO. 160

January 1979

This report was prepared as part of the documentation for the S-l programming system, under a subcontract from Lawrence Liver-more Laboratory to Stanford University, Computer Science Department, Principal Investigator Professor Gio Wiederhold, Contract No. LL L P09083403. Other Lawrence Livermore Laboratory as well as Advanced Research Projects Agency contracts have supported the facilities at the Stanford Artificial Intelligence Laboratory, which was used in the execution of this work. The S-l project is supported at Lawrence Livemore Laboratory of the University of California by the Department of the Navy via ONR Order No. N00014-8-F0023.

STAN-CS-79-714

PCFORT

A Fortran-to-Pcode Translator

Fernando Castaneda
Frederick Chow
Peter Nye
Dan Sleator
Gio Wiederhold

TECHNICAL REPORT NO. 160

January 1979

COMPUTER SYSTEMS LABORATORY
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305

This report was prepared as part of the documentation for the S-1 programming system, under a subcontract from Lawrence Livermore Laboratory to Stanford University, Computer Science Department, Principal Investigator Professor Gio Wiederhold, Contract No. LLL P09083403. Other Lawrence Livermore Laboratory as well as Advanced Research Projects Agency contracts have supported the facilities at the Stanford Artificial Intelligence Laboratory, which was used in the execution of this work. The S-1 project is supported at Lawrence Livermore Laboratory of the University of California by the Department of the Navy via ONR Order No. N00014-8-F0023.

PCFORT

A Fortran-to-Pcode Translator

Fernando Castaneda, Frederick Chow, Peter Nye, Dan Sleator,
and Gio Wiederhold

TECHNICAL REPORT NO. 160

January 1979

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305

ABSTRACT

PCFORT is a compiler for the FORTRAN language designed to fit as a building block into a PASCAL oriented environment. It forms part of the programming systems being developed for the S-1 multiprocessor. It is written in PASCAL, and generates P-code, an intermediate language used by transportable PASCAL compilers to represent the program in a simple form. P-code is either compiled or interpreted depending upon the objectives of the programming system.

A PASCAL written FORTRAN compiler provides a bridge between the FORTRAN and PASCAL communities. The implementation allows PASCAL and FORTRAN generated code to be combined into one program. The FORTRAN language supported here is FORTRAN to the full 1966 standard, extended with those features commonly expected by available large scientific programs.

KEY WORDS: FORTRAN, S-1, PASCAL, P-code, translator, compiler

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

ACKNOWLEDGEMENT:

We wish to acknowledge crucial support for this work which has been received from the Department of the Navy via Office of Naval Research Order Numbers NO001 4-76-F-0023, NO001 4-77-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-03-0516), and from the Stanford Artificial Intelligence Laboratory (which receives support from the Defense Advanced Research Projects Agency and the National Science Foundation).

We also would like to acknowlege the invaluable assistance of Erlk Gilbert, Curt Widdoes, and David Fuchs.

1. Introduction

The FORTRAN compiler described in this document, PCFORT, was written specifically to serve in a PASCAL environment [JeW78], using P-Code as an intermediate pseudo machine [NAJ75]. 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]. Ail aspects of FORTRAN which are commonly used in large scientific programs are available, including such features as SUBROUTINES, iabelled 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.

7.7 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 additional 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 a blank. 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. PCFORT 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 ($tudent) 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 P-Code, does no explicit optimization of generated code. Recognition of common subexpression, for instance, will require at least an additional pass in a compiler, Current research in the PASCAL/P-Code project at UCSD may lead to such an optimizer operating on P-Code. The compiler is also not aware of the register structure in the underlying machine. It is the function of a P-Code compiler (e.g., SOPA[wag78]) or a P-Code interpreter to carry out the requested P-machine actions in a manner which utilizes the underlying hardware effectively.

The 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-l project documentation note [giw77]. In our case the P-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 PCFORT compiler Itself, but is of major concern when transporting FORTRAN application programs, which manipulate characters, between computers, since FORTRAN standards has 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 however since this package is written In PASCAL. The P-Code generated by our PASCAL compiler is combined with the P-Code from PCFORT prior to translation to machine code. 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 PCFORT, the compiler and the run-time package, are of course constrained due to the facilities provided by the PASCAL P-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.

Another aspect of the P-code environment is that it does not provide well for separate compilation of routines. The stack orientation of the P-Code machine further inhibits external procedures. PCFORT will 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 P-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 [kew78].

1.2 Conclusion

The PCFORT FORTRAN compiler is a building block within a PASCAL and P-Code environment, which can take care of existing needs for the continued use of FORTRAN coded algortthms. 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 PCFORT, and specifies in detail any differences with 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 PCFORT FORTRAN in comparison with standard FORTRAN compilers, and especially in comparlslon with the FORTRAN '66 Standard [ANS66]. Most of the limitations are expected to be temporary. The background for the limitations is given later in the section.

2.1 Statements

The following FORTRAN statement types have been implemented:

Declaration statements:

DIMENSION
COMMON
EQU I VALENCE
IMPLICIT
EXTERNAL
LOG I CAL
INTEGER
COMPLEX
REAL
DOUBLE PRECISION
DATA

Executable statements:

The assignment statement
ASS I GN
IF (logical and arithmetic)
GOT0 (unconditional, computed, and assigned)
CALL
RETURN
PRINT
STOP
DO
READ
WRITE
REWIND

Other statements:

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

Not implemented:

END FILE
BACKSPACE
PAUSE
ENTRY

2.2 Program Format

Some restrictions on program FORMAT are imposed by PCFORT:

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.").

Single quotes within a string can only appear in Hollerith strings (e.g., 5HDON'T). If two quotes appear in sequence, the string is considered as two strings when scanned by the lexer.(e.g.'DON''T' is divided in strings 'DON' and 'T').

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.

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 can be the same as a variable 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(4HFOURF8.516))
and     (X,3X,X,'ONE',X,/,X,2(4HFOUR,F8.5,I6))
```

are equivalent.

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 usual naming conventions are used to determine if a variable or function is of type integer or real, but they may also be explicitly declared. The naming conventions **may also** change through the use of an IMPLICIT statment.

The following **precisions** are possible:

INTEGER and LOGICAL: quarter word, half word, single word, double word
default: single word

REAL: single word, doub e word: half word not yet implemented
default: single word

COMPLEX: two single words, two double words
default: two single words

Precisions are specified in quarter words, **as** in IBM FORTRAN:

```
INTEGER*1 AAA
LOG I CAL*8 BBB
COMPLEX CCC
COMPLEX*16 FUNCTION DDD
REAL*8 EEE or DOUBLE PRECISION EEE
```

Automatic conversion occurs between and among any precision of integer and any precision of real, (Reals are converted to integers by truncation.) Any other conversions must be done explicitly using standard conversion functions,

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.

Exponentiation of a complex by an integer is allowed.

2.3.2 Constants

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

The upper limits allowed for integers are 255 for quarter-word integers, 13107 1 for half-word integers, 34369738367 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 1.701411843E+38 and 1.469368010E-39 respectively, for all precisions.

Currently, double precision constants are not recognized by our P-Code. For the time being, they are converted to single precision by PCFORT.

2.4 Arrays and Storage Management

Array *subscripts:*

Array subscripts may consist of any legal integer expression.

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

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

The specification of array elements in DATA and EQUIVALENCE statements with only one dimension for arrays of several dimensions is accepted. (e.g. For an array dimensioned as A(3,3), the array element A(2,3) may be specified as A(6)).

Arrays with adjustable dimension:

No restriction is made on the value of an actual argument that represents the dimension 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.

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 labeled common may not require a storage area larger than the amount specified by the first declaration of the common, as in the following example:

wrong:

```
COMMON /X/ A
DIMENSION A(20)
END

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

right:

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

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

One way to go around such a problem is to use the switch that fixes a minimum size for the common areas. With it, if the area required in the first declaration is smaller than the one declared the second time, the switch should be set to the space needed for the larger one.

Storage allocation:

No assumptions 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 word on half-word boundaries, single word in 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 initialization and DATA statement are formed as follows:

```
Type*s a*s1(k1)/x1/,b*s2(k2)/x2/,. . .,z*s3(k3)/x3/

DATA a(k1),... ,d(k4)/x1/,e(k5),. . .,h(k8)/x2/,. . .
```

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

*s,*s1,*s2,... are optional, each s represents one of the permissible length specifications for its associated type;

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

(k1),(k2), ... 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 implied;

xl, x2, . . . are constants or lists of constants. /x1/,/x2/,/x3/ . . . are optional in **a** declarative statement, and are used to specify initial values for single variables and array names. In **a** DATA statements, they are not optional, and specify initial values for the preceding list of variables, array elements or array names;

2.5.1 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. In initialization with boolean constants, however, only the first quarter word of the location is tempered with,

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. 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 variable is taken as two real variables, and they correspond to two initialization constants. The enclosing parentheses are not allowed.

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 value 3.2 is going to be used 6 times).

4. Function names or subprogram parameters cannot be Initialized.

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

2.6.2 Initialization with character strings

The initialization of variables with 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.

6. 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.

2.6.2.1 Examples

The following example:

```
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'/
```

will **cause** the following initialization:

```
VARIABLE    VALUE

   M        'ABCD'
 A(1)       'ABCD'
 A(2)       'EFGH'
 D(1)       unintial ized
 D(2)       'AB'
 D(3)       'CO'
 C(1)       'ABCD'
 C(2)       'EFGH'
 C(3)       'I'
 E(1)       'ONEI'
 E(2)       'TWO'      ;before this, E(2) contained 'SMORE' but 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 I ist, thus it is not
 F(1)       'MOM'        overwritten
 F(2)       'MOM'
 F(3)       'MOM'
```

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

2.6 Subprograms

The restrictions with regard to subprograms are:

Functions:

Function parameters (i.e. CALL TRIG(SIN,X,Y)) are currently not allowed, since the PASCAL P-code combination we are using does not permit them.

A statement function must have at least one argument. A function with no parameters 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. Exceptions to this are constants and expressions as actual parameters.

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.

ENTRY statement:

The ENTRY statement, available in many compilers, although not part of the standards, has not been Implemented. The way to implement it in a PASCAL envlronment is not yet established.

2.7 User Options; the SET statement

User options are indicated by setting various flags. Currently there are two: BCHK turns on array boundary checking, and CSIZ specifies a minimum size for the common area following the SET statement. A flag may be set to T, F or an integer value, For example:

```
SET BCHK=T,CSIZ=1200
```

SET statements may appear anywhere in a program. The defaults are F for BCHK, and 0 for CSIZ.

CSIZ only applies to the common areas that appear for the first time in the next COMMON statement following the occurence of the option. It is reset to its default value at the end of each COMMON statement and at the beginning of each program unit.

2.8 Input/Output

2.8.1 File handling

PCFORT 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 25 are given internal representations of FILEO, FILE2, FILE3, ... FILE26. The mapping between these pseudo-files and actual devices or disk files is done at runtime, usually by a direct prompt at the terminal. Example:

```
FILE1? DATA1
FILE2? 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. (The latter is the case for the current S-l run-time [gwa78]). 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 file. 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.

The BACKSPACE and END FILE statements are not implemented.

2.8.2 The READ and WRITE statements

The standard READ, WRITE and FORMAT statements use FORTRAN run-time routines. These routines are currently stored in P-Code form and copied to the end of the main P-Code file when necessary, and have to be compiled together into the machine code in a program that uses these statements.

Both formatted and unformatted reads and writes are handled. Unformatted write uses 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 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 to an A-formatted field. It Is to be warned that 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.

2.8.3 The PRINT statement

The READ, WRITE, and FORMAT statements use Fortran run-time routines which are currently stored in P-code form and copied to the end of the main P-code file when necessary. This adds substantially to the time required to translate the P-code file. This may be bypassed by using the PRINT statement, which makes use of PASCAL run-time routines, and acts somewhat like a Pascal WRITELN statement. 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 a&ding 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 standard file OUTPUT unless a file number is added, preceded by a colon ("PRINT:2"). In this case, the file must be first opened using an OPEN statement ("OPEN2").

Here are some examples:

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

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

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

COMPLEX*8 X
PRINT 'THE ANSWER IS', X*(2.,0.):10
  result: THE ANSWER IS 2.0       0.0

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

2.9 Miscellaneous

DO statement:

A DO statement must have an integer or integer variable for its upper bound and step size. Hence, "DO 30 I=1,J+1" is illegal. An integer expression may be used as the lower bound. The control variable may 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. I.e. the usage of an integer variable is not restricted with regard to whether it got its value by regular integer assignments or by the ASSIGN statement for statement labels. An array element can be used for the variable.

3. Overall Organization

3.1 Structural Scheme

PCFORT'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 PCFORT is' based on these modules. It is structured according to the relationships among the various modules, Despite its length (about 9000 lines), PCFORT easily presents itself once its structure is revealed.

Basically, when the compiler processes a given program statement, It either generates code from it or remembers the information given in the text by building some internal structure, which invariably is a linked list of a particular type, A module in PCFORT can guarantee its own existence only if it 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 different types of statements. 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.

(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 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. “DOUBLE PRECISION”. 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(I,4)".

3.2 Error Handling

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

(1) Errors discovered while scanning a program statement: PCFORT 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, PCFORT will try to generate extra dummy code to make the code already generated for the statement acceptable by the P-Code translator. PCFORT will continue as usual 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 first lexeme it processes and exit with LXC pointing to the one after the last lexeme it processes.

Warnings are output when errors are discovered in the program which PCFORT 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 PCFORT. The resulting behaviour of the program is easily predictable by the user.

PCFORT always prefers warning instances to error instances. I.e. for each user error, PCFORT 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 iexeme.

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 *Lexeme 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.
ANDOP, OROP	.AND., .OR.
NOTOP	.NOT.
REALCON	a FORTRAN real constant (not including preceding sign).
DPCON	double precision const (not including preceding sign).
I NTEGERCON	an integer constant (not including sign).
STRINGCON	quoted or Holierith constant
TRUECON	.true.
FALSECON	.false.
IDENTIFIER	a sequence of characters, the first of which must be letter and the rest may be letters or numbers
EXPLMARK	!
QUOTMARK	
NUMSIGN	#
DOT	
DOLS I GN	$
PERCENT	%
AMPERSAND	&
COLON	
SEMI COLON	
LESSSIGN	<
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 blanks where the previous statement was. 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. COL1 TO6 is printed in the listing file and the comment itself is read into the listing file (procedure SKIPLINE). The variable LINEN0 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 tines 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 spaces 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 also written to the P-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

The array LEXEME is. then fitted with iexemes that are recognized through a case statement based on the first characters of the iexemes 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 iexeme 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 function DIGITSTRING (which will, in this case, always return TRUE, since we already know it is a digit string), finds the last digit. if the digit' string is followed by an H, then the lexeme is a Holierith 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 iexemes being processed in a single pass: the integer and the dot-word). If the dot is not followed by a letter, DIGITSTRING is catted 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 iexeme is an integer.

if the first character is a dot, then the iexeme 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. A string like 'ab''cd' is separated into two iexemes of type string ('ab', 'cd'). 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 parenthesis, 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).
```

5. Statement Classifier

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

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

                   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 GOT0 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 only 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 will 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

call INITCOMPILER to initialize everything
call BLOCK to process the main routine
generate a return and a label that indicates how much storage is needed for the main block.

while there are more subprograms do

    call FUNC_ STMT or SUBR STMT to get type and arguments of subprogram
      or call BLKDATASTMT if BLOCK DATA statement

    call BLOCK to process body of subprogram

call VARINITIALIZATION to generate code to intialize any variables that should be initialized
copy run-time routines to end of P-Code file if they are needed

6.2 Procedure Block

call INITBLOCK (see Section 19.1)
call LEXER to get statement
set global lexeme pointer, LXC, to point to first lexeme
call CLASSIFY to find out what kind of statement it is

while there are more declaration statements, FORMAT or SET statements do

    call the appropriate routine to process it
    call LEXER to get statement
    set global lexeme pointer, LXC, to point to first lexeme
    call CLASSIFY to find out what kind of statement it is

call STORAGE ALLOCATION to allocate storage for the variables that have been declared
call FILL_ADDRESS_INITIALIST to copy these addresses into the list of variables to be initialized

while there are morestatement function declarations, FORMAT or SET statements do

    call STMT_FUNCTION, FORMAT_STMT or SET_STMT
    call LEXER to get statement
    set global lexeme pointer, LXC, to point to first lexeme
    call CLASSIFY to find out what kind of statement it Is

if we are not processing a BLOCK DATA block, generate SST and ENT instructions

whlle statement is an executable statement, FORMAT or SET statement do

    if there is a FORTRAN label, then enter it in the label table if it is not there already and generate code for a P-Code label (ENTERLABEL)
    call the routine to process the statement
    if we are not about to process statement within a logical IF statement then do
        if we have been processing an IF statement, then generate the P-Code label to be jumped to if the condition was false
        if there is a FORTRAN label and it is an ending for a DO loop, then generate the appropriate code
        get the next statement

if the END statement we are now on has a label, enter it in the label table, generate the corresponding P-Code label and give a warning.
check if any do loop is still open
Issue warnings If any labels or variables have been used only on the left-hand side or only on the right-hand side
If the block is not main or a block data, generate a return and a label that indicates how much storage is needed for this block.
get the next statement

7. Symbol Tables

7.1 The structure of the tables

There are five symbol tables:

1) The main symbol table (SYMBOL) keeps track of variables, subprogram names and FORMAT labels within a single unit (main program or subprogram).

2) The EXTNAME table keeps track of subprogram names throughout all the program units.

3) The LABELNO table keeps track of FORTRAN labels within a single program unit.

4) The COMNAME table keeps track of common areas.

6) The STDFUNCTABLE contains the name 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 unit. The other three are cleared only once, at the beginning. The storage used by the cleared entries is automatically reclaimed through the garbage collection facility in the PASCAL in which PCFORT 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 if it is not already there, and then adds any information to the symbol 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 proper table, The following description of how FSYMBOL works, therefore, is applicable to the other three routines.

When FSYMBOL Is called, it calls routine BUILDSYMBOL with a name and a pointer to the head of the table as parameters. BUILDSYMBOL looks for an entry in the table with that name by calling procedure SYMLOOK. IF SYMLOOK finds the name in the table, it sets FOUND to TRUE and returns a pointer to the symbol in SPTR. If it does not find the symbol, it creates a new record, calls CLEARSYMBOL to set the default values of the record, sets FOUND to FALSE, and returns the pointer to the new record in SPTR. If FOUND Is false, the BUILDSYMBOL knows that the record is a new record and inserts the implicit type of the

symbol, and then passes SPTR back to FSYMBOL. 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: 1) to retrieve the pointer to the symbol table entry, 2) to assert information about the symbol as given in the parameters in the call, and 3) to test the properties of the symbol against the values given in the parameters in the call. Each of the routines depart from 3) somewhat, and the details are given in their sections following.

7.3 The main symbol tab/e

The main symbol table stores information about the characteristics of the variables in a block, the most important of which is their addresses. it also stores the FORMAT labels. A space in memory for saving the address of the FORMAT string is allocated for each FORMAT label (see Section 26).

It uses records-of type SYMBOL:

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

FUNCTYPE = (NOTEXTERNAL,EXTERNAL,EXTSUBR,EXTENTRY,EXTFUNC,STMTFUNC,
            INTRINSTDEXT):

SYMBOL = PACKED RECORD
           LSON,RSON: ↑SYMBOL;        (* POINTERS TO SONS *)
           NAME: THENAME;             (* SYMBOL NAME, 6 CHARACTERS LONG *)
           STYPE: DATATYPE:           (* THE TYPE OF THE VARIABLE; I T SHOULD
                                         BE SET TO NONE IF SUBROUTINE NAME *)
           WHEREDEFINED,              (* PROGRAM LINE NUMBER IN WHICH
                                         VARIABLE APPEARS THE FIRST TIME *)
           LEVEL,                     (* ADDRESSING LEVEL FOR THE VARIABLE *)
           ADDRESS: INTEGER;          (* -1 IF NOT YET ESTABLISHED. *)
           USED_LHS,                  (* TRUE IF VARIABLE WAS GIVEN A VALUE *)
           USED-RHS,                  (* TRUE IF VARIABLE'S VALUE WAS USED *)
                                      (* ABOVE 2 NOT USED IF FORMATLABEL, EXTERNAL,
                                         SUBROUTINE, STANDARD FUNCTION OR
                                         EXTFUNC NOT USED AS FUNCTION VARIABLE *)
           S-EXPLICIT: BOOLEAN; (* TRUE IF TYPE EXPLICITLY DECLARED *)
  CASE S_FUNCSUBR: FUNCTYPE OF  (* NOTEXTERNAL IF NOT EXPLICI TLY ASSERTED *)
       INTRINSTDEXT:(PTRSTD:↑STDFUNCTABLE);(* POINTER TO STANDARD FUNCTION
                                              TABLE IF STARDARD FUNCTION NAME *)
       STMTFUNC:      (SEGMENNUM: INTEGER);   (* SEGMENT NUMBER OF ITS P-CODE
                                                 PROCEDURE BLOCK *)
       NOTEXTERNAL; (S1_EQUIVALENCE,          (* TRUE IF VARIABLE EQUIVALENCED*)
                     S2_EQUI VALENCE,         (* USED TO INDICATE IF AN EQUIV.
                                                 VARIABLE HAS BEEN PROCESSED IN
                                                 STORAGE ALLOCATION TO CHECK
                                                 EQUIVALENCING TWICE *)
                     S-COMMON,                (* TRUE IF COMMON VARIABLE *)
                     S-DUMMY,                 (* TRUE IF DUMMY ARGUMENT *)
                     INITIALIZED: BOOLEAN:    (* TRUE IF VARIABLE IS
```

```
                                                INITIALIZED, FALSE OTHERWISE *)
                (* FOLLOWING FIELDS DO NOT HAVE CORRESPONDING PARAMETER
                   IN PROCEDURE FSYMBOL *)
                PTRCOM:↑COMNAME;          (* POINTER TO THE COMNAME TABLE,
                                             USED ONLY IF COMMON SYMBOL *)
                DIMENSION: INTEGER:       (* 0 IS THE DEFAULT DIMENSION
                                             IF NOT EXPLICITLY DIMENSIONED *)
                S_CON:ARRAY[1..MAXDIM] OF BOOLEAN: (*TRUE IF THE ITH
                                             DIMENSION IS CONSTANT *)
                DIMEN:ARRAY[1..MAXDIM] OF DIM: (* EITHER THE CONSTANT
                                          DIMENSION OR THE POINTER TO THE SYMBOL
                                          TABLE ENTRY IF VARIABLE DIMENSION *)
        END:
```

Its main procedure, FSYMBOL, has parameters that correspond to the record fields:

```
PROCEOURE FSYMBOL (VARSPTR: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. It is necessary 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.

7.4 The label number tab/e

Both statement labels and FORMAT labels are entered into this table. For each statement label, It also stores the P-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=I ") or the right-hand side ("GOTO100"), 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,                          (* FORTRAN LABEL *)
            PLABEL: INTEGER;               (* PCODE LABEL NUMBER ASSOCIATED *)
            LSON, RSON: ↑LABELNO;
            IS_ON_RHS,
            IS_ON_LHS: BOOLEAN;            (* TRUE IF THIS LABEL NUMBER HAS OCCURREO
                                              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 FLABELNQ (VAR LPOINTER:POINTLABELNO;
                    NUMBER:INTEGER;        (* FORTRAN LABEL *)
                    LIS_ON_RHS,
                    LIS_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 GOT0 and arithmetic IF statement processors, the DO statement processor and the READ/WRITE statement processor.

7.5 The common tab/e

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 = PACKED RECORD
            LEVEL,                         (* PSEUDO LEVEL NUMBER FOR THIS COMMON
                                              AREA *)
            LENGTH, STADDR: INTEGER;       (* LENGTH OF THE COMMON BLOCK IN QUARTER
                                              WORDS AND STARTING ADDRESS *)
            PTRCOMLIST: CCOMLIST;          (* POINTER TO THE LINKED LIST OF COMMON
                                              ELEMENTS IN THIS AREA *)
            LSON, RSON: ↑COMNAME;
            NAME: THENAME;                 (* NAME OF THE COMMON AREA *)
          END;
```

and accessed by the routine FCOMNAME during storage allocation:

```
PROCEOURE FCOMNAME (VARCPOINTER: POINTCOMNAME;
                    CONAME: THENAME);
```

LEVEL Is filled automatically inside CLEARCOMNAME, Immediately after the entry Is created, In such a way that each common area has associated a different level number If the switch VARCOMMON is on, or level 2 if it Is off (see Section 10).

PTRCOMLIST, which points to a linked list of variables, is built when processing COMMON declarations. At the beginning of each program unit, the field PTRCOMLIST of all entries is cleared.

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 tab/e

The external name table keeps track of the existence and calls of the various subprograms. A symbol can be in the EXTNAME table and in the SYMBOL table at the same time. In this case, the symbol is either used as an external subprogram name in the program unit, or 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 variable.

A symbol is inserted in the external table when it is called or defined. This occurs in 1) procedure USERFUNC, which processes calls, 2) the FUNCTION statement processor and. 3) **the** SUBROUTINE statement processor.

The table is made up of records of type EXTNAME:

```
EXTNAME = PACKED RECORD
            LSON, RSON: ↑EXTNAME;
            NUMBER : INTEGER; (* SEGMENT NUMBER ASSOCIATED TO THIS
                                 SEGMENT NAME ENTRY *)
            XFUNCSUBR: FUNCTYPE; (* MUST BE ONE OF EXTFUNC, EXTSTMT,
                                    EXTENTRY *)
            TYPEEXPLICIT,        (* TRUE IF EXPLICIT TYPE IN SUBPROGRAM
                                    HEADING *)
            IS_DEFINED,          (* A SUBPROGRAM BLOCK EXISTS FOR IT *)
            IS-CALLED: BOOLEAN: (* INVOKED AT LEAST ONCE *)
            STYPE:DATATYPE;      (* THE TYPE OF THE FUNCTION; IF
                                    SUBROUTINE, THIS FIELD NOT USED *)
            NAME: THENAME;
        END:
```

and accessed by the routine FEXTNAME:

```
PROCEOURE FEXTNAME (VAR EPOINTER: POINTEXTNAME:
```

```
                EXNAME:THENAME;
                EXTYPEEXPLICIT: BOOLEAN; (* TRUE IF EXPLICIT TYPE IN
                                            SUBPROGRAM HEAD I NG *)
                EXTYPE:DATATYPE;          (* 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 has associated 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 FSYMBOL is called from 1), 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 FSYMBOL is called from 2), 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 STDFUNCTABLE. It has the following type of record:

```
STOFUNCTABLE = 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

A variable can have any of the following datatypes:

```
DATATYPE = (NONE,          (* NONE OF THE OTHER TYPES *)
            LOGICAL1,      (* EQUIVALENT TO THE TYPE BYTE *)
            LOGICAL2,      (* LOGICAL HALF WORD *)
            LOGICAL4,      (* LOGICAL SINGLE WORD *)
            LOGICAL8,      (* LOGICAL DOUBLE WORD *)
            RE8,           (* REAL DOUBLE PRECISION *)
            RE4,           (* REAL *)
            RE2,           (* REAL HALF WORD *)
                  (* THIS TYPE IS NOT YET FULLY IMPLEMENTED. THE
                     COMPILER WILL RECOGNIZE IT BUT NO CODE CAN BE
                     GENERATED FOR IT YET, *)
            INT8,          (* INTEGER DOUBLE WORD *)
            INT4,          (* INTEGER SINGLE WORD *)
            INT2,          (* INTEGER HALF WORD *)
            INT1,          (* INTEGER QUARTER WORD,
                              EQUIVALENT TO THE TYPE CHAR *)
            COMP8,         (* COMPLEX *)
            COMP16,        (* COMPLEX DOUBLE PRECISION *)
            FORMATLABEL (* A FORMAT LABEL HAS THIS TYPE WHEN INSERTED
                           IN THE SYMBOL TABLE *)
```

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 will occur In procedure STORAGE-ALLOCATION (see Section 12, Storage Allocation).

8.1 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 the star 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 the variable if it is an array, This procedure returns the number of elements in the array in its reference parameter ITEMS.

If the variable is initialized, procedure VARINIT is responsible for the steps involved. This procedure builds a list of the variables to be initialized. This list will be formed for the INITIALIST records (see Section 8.7, Data Statement). The root of the list Is the global variable HEADINIT. An entry in the list is created for each element to be initialized. This means that a simple variable will have only one entry in the list, but an array of 6 elements will have 6 entries in the list; a single complex variable will have 2 entries in the list, the first one for the real part and the second one for the imaginary part.

The rules for initializing variables are described in Section 2.6, Initializing Variables.

VARINIT is entered with variable ITEMS set to the number of elements that are going to be initialized and 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 ITEMS. Procedure FILL_VALUES is then called which traverses the lexemes with the initialization values and fills them in the fields in the nodes just created. In this process, it uses procedure INSERT_VALUE.

Procedures EXTEND LIST, FILL_VALUES and INSERT VALUE are also used in processing the Data statement. See Section 8.7, The Data Statement, for more detailed descriptions.

The same initialization list is used for all the program units of a program, lengthening as more initializations are specified, but the symbol table for each block is cleared at the beginning of a new block. For this reason, the address of the variables to be Initialized have to be saved in INITIALIST. This is done in procedure FILL ADDRESS_INITIALIST which is called after the storage allocation has taken place (see Section 9.1, Procedure FILL_ADDRESS_INITIALIST).

8.2 Dimension Declaration

Procedure DIMENDECL scans and processes the FORTRAN DIMENSION statement. The symbol table entries for the variables are updated with the dimension information. It uses procedure ISARRAY to obtain the dimension information.

8.3 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.

8.4 Common Declaration

Procedure COMDECL scans and processes a common declaration. The common name table Is built inside this procedure and lists with the common variables in each common area are made. 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 the llst of common variables for each common areas 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 list for that area, Using this pointer, the new 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 reasonfor 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.

8.5 Equivalence Declaration

Procedure EQUIVALDECL scans and processes the equivalence declaration. This procedure builds the list of equivalence groups and it also builds the circular lists of equivalent variables that form the equivalence groups. A pointer to the beginning of the list of equivalence groups is stored in EQUIVHEAD.

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 HIGER BOUNDS
                                   OF THE EQUIVALENCE GROUP *)
            LEADER: ↑EQLIST;    (* POINTS TO FIRST ELEMENT IN LIST OF
                                   EQUIVALENCES VARIABLES THAT FORM GROUP *)
            NEXT: ↑EQGROUP;     (* POINTS TO NEXT GROUP *)
            ALLOCATED,          (* TRUE IF THE GROUP 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 EQUIVALENCED VAR. *)
           DIMENSION:ARRAY[1..MAXDIM] OF INTEGER;
                (*USED TO STORE THE COORDINATES OF ARRAY ELEMENT
                  EQUI VALENCED *)
           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 EQUIVALENCE0
       I N ONE EQUI VALENCE 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 equivalcnced 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.

8.6 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 only when the external symbol is called.

8.7 The DATA Statement

In most FORTRAN compilers, DATA statements are handled by setting up the binary load file so that the locations which are specified by the variable to be initialized are loaded with the initial values at the time the program is loaded. It is not possible to do this in P-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.

Procedure DATA STMT scans and processes a DATA statement. The initialized variables are inserted% a list of the variadies to be initialized at the beginning of program . execution. The generation of code for the actual initialization of variables is done in procedure VARINITIALIZATION , described in Section 9.2.

The global variable HEADINIT points to the head of the list of variables to be initialized. The variable TAILINITLIST points to the last element The list is formed with records called INITIALIST with the following structure:

```
INITIALIST = PACKED RECORD
                SYHTABPTR:↑SYMBOL;   (* POINTER TO SYMBOL TABLE ENTRY
                                        OF VARIABLE TO BE INITIALIZED *)
                LOCSIZE: INTEGER:    (* SIZE OF INITIALIZE0 LOCATION:
```

```
                                        FOR COMPLEX, SI ZE OF EACH HALF *)
    NEXT: ↑INITIALIST;          (* NEXT NOOE *)
    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 PREVI OUS NOOE, USED I N
                                   INITIALIZATION WITH STRINGS *)
CASE AHOUNTYPE:LEXTYPE OF       (* LEXTYPE OF THE STRING VALUE *)
STR I NGCON:
    (STRLEN: INTEGER);          (* IF INITIALIZATION WITH STRING,
                                   LENGTH OF THE STRING CONSTANT *)
INTEGERCON,REALCON,DPCON:
    (NEGATIVE:BOOLEAN);         (* TRUE IF CONSTANT IS -VE *)
END;
```

A group Is formed by ail the variables that appear before the two slashes that surrounds the initial values for that group. For each group of variables to be initialized, procedure DATA_STMT adds nodes for the variables to be initialized in **INITIALIST** by calling **procedure FORM** VAR_LIST. The list is then updated with the initial values for the variables just Inserted-by **calling** procedure FILL_VALUES. Variable **FIRST IN LIST** is returned from FORM_VAR_LIST pointing to the **first element** of the group **just inserted** and Is used by **FILL_VALUES to tell** where to start entering the initialization values.

Here is a more detailed description of the procedures used:

Procedure FORM_VAR_LIST 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 INITIALIST for these variables by calling procedure EXTEND-LIST.

One entry is created for a simple variable. Complex variables are inserted in the list of initialized variables as two reals: 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 initiaiizatlon specification.

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 variable, 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 **INITIALIST** 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 INITIALIST while saving the values in the AMOUNT field of INITIALIST. For each value, this procedure gets the number of times the value is repeated, INSERT VALUE Is then called this number of times. Fields NEGATIVE or STRLEN of INITIALIST are set directly in FILL_VALUES depending on the type of the constant. 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 inserting the iextype and the amount expressed in characters.

The procedures EXTEND LIST, FILL VALUES and INSERT VALUE are also used in processing Initiailzations in type-specific declaration statements.

9. Initialization of Variables

The Initialization of variables requires three steps, First, a list of the variables to be initialized is formed during the processing of type-specific declarations (Section 8.1) and DATA statements (Section 8.7). In the second step, the addresses of the variables to be initialized are saved in the LEVEL and ADDRESS fields of the record entries in INITIALIST when procedure FILL ADDRESS_INITIALIST is called after storage allocation has occurred. Finally, code are generated for the initializations at the end of compilation by calling procedure VARINITIALIZATION. These last two procedures are described in this section.

9.1 Procedure FILL_ADDRESS_INITIALIST

This procedure finds the address of a variable once storage has been allocated to it and enters it 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, All 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 value of the displacement 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 previously store 0. Field LEVEL is obtained directly form the LEVEL field in the symbol 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.

9.2 Procedure VARINITIALIZATION

This procedure is called by the main procedure after all the program units are compiled. It generates code for the initialization of variables and the loading of FORMAT Ispecifications into memory at execution time, the latter being done by calling procedure INIT_FORMATS (see Section 26.2, Initialization of Formats),

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

Procedure VARINITIALIZATION first generates code for the head of the special procedure $INIXX by calling procedure BLKCODE_GENERATION. Then, it generates code for the body of procedure $INIXX. This consists of a series of LDC-STR P-Code instructions that will 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 P-Code instructions.

Before generating code for the return of procedure $INIXX, procedure VARINITIALIZATION calls procedure INIT_FORMATS that generates code for the loading of the FORMAT string specifications into memory. After this, procedure $INIXX Is closed with the RET and DEF P-Code instructions.

10. Storage Allocation Structure

10.1 The problem

In P-Code, there are a number of static /eve/s, each of which may have one or more procedures associated with it. Each procedure has a set of local variables associated with it. When a procedure is entered, space for its variables is allocated; at exit, the space is deallocated. Thus, the values of ail of the local variables of a procedure are undefined when that procedure is entered.

In FORTRAN, however, all of the variables of each subroutine 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 all of these variables must be allocated at the beginning of the program, even though some of them may only be accessed when certain subroutines are entered.

in P-Code terms, this means that all variables in a FORTRAN program must be on some level that is lower than or the same as the level of the main program.

The fact that PCFORT is one pass has an important ramification: the total amount of storage needed for the main variable level and for each of the COMMONs is unknown until all the code for all the procedures and subprograms has been emitted. This presents two problems:

1. SOPA, the P-code compiler currently used by PCFORT, demands that the amount of storage needed for the local variables of any one procedure must be indicated before the emission of P-Code for the next procedure; i.e. the amount of space needed for the variables of the main routine must be output before generating any of the code for any of the subroutines.

2. If the main variables and common variables are on the same level, the address of any variable following those declared to be in a common area cannot be definitively determined until the size of that common area is known, For example, in the following, if the address of B is fixed before subroutine ZZZ is processed, there will not be enough room in common X for array C:

```
COMMON/X/ A (10)
B = 1.3
END

SUBROUTINE ZZZ
COMMON/X/ C (20)
END
```

11.1 Partial Solution

A partial solution to this problem involves assigning to the blank common area its own level and restricting the rest of the common areas by specifying that the first time a common area is declared in a program unit, its size is the larger this area can have in any other program unit that appear later. Then the levels are distributed as follows:

Level 1 -- the blank common area (dummy procedure)

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

Level 3 -- all other variables (dummy procedure)

Level 4 -- main block (no variable)

Level 5 -- all subprograms (no variable)

Level 6 -- all statement functions (no variable)

This scheme is advantgeous because of the fact that P-Code does not require that procedures be In any specific order. Thus, the code for the "procedures" In levels 1 - 3, which Includes how much storage is needed for these procedures, can come after the cods for levels 4 through 6.

To make this work, there must be a series of procedure calls at the beginning of the program, each of which allocates storage for that level and then calls the procedure for the next higher level. Here is a PASCAL representation of the idea:

11.2 PASCAL representation

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

       procedure FORMAIN;

          procedure USERSUBROUTINE;

             function STATEMENTFUNCTION (real X);

               begin
               STATEMENTFUNCTJON := 2*X;
               end:

             begin(* USERSUBROUTINE *)
             k := 2.0;(* normal variable *)
             i111 := 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 genera l var area *)
       FORMA1 N;
       end;

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

  begin (* dummy for blank common area *)
  GENCOMMON;
  end:
```

11.3 The CMN instruction

This scheme only partially solves the commons problem, as the size of any of the commons in level 2 must be known before another common is declared.

A complete solution that has been proposed is to have a new P-Code instruction, called CMN, which would assign to each named common area a pseudo-level number (level 9 and above). This pseudo-level would represent a special loader segment, which would be pointed to by a register. We have anticipated this solution by including a switch, VARCOMMON, which will emit this new instruction when set to TRUE.

To minimize the user frustration before this is Implemented, the current implementation provides a user switch called CSIZ that allows the user to indicate the size of the common directly.

12. Storage Allocation

This procedure 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.

Each variable is assigned a level number and an offset. Functions and subroutine names and their dummy arguments are assigned the level number of 5 . Statement functions and their dummy arguments are assigned the level of 6. Common variables are assigned levels 1 (blank common area) and 2 (rest of the common areas) respectively if switch VARCOMMON is not set, If the switch is set, each common area except the blank common is given its own pseudo-level, beginning at level 9 (see Section 1 1.3). All other variables are assigned a level of 3.

The allocation of space is done using a global variable called DISPLACEMENT that keeps track of the space already allocated on level 3. About 600 quarter words of storage are needed by the run-time routines, and DISPLACEMENT is initialized to point to the first free position after that. Every time a space for a variable is needed, DISPLACEMENT is adjusted, if necessary, to lie on a half, single or double word boundary. Its value is then stored in the field ADDRESS of the symbol table. It is then incremented by the proper amount.

The space is allocated in a specific order:

1) Common variables and equivalenced groups containing a common variable. 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 (complex) word variables at the next double word boundary.

Common variables are passed to the STORAGE_ALLOCATION routine in the form of a list (see Section 7). The head of the list is stored in the PTRCOMLIST field of the common name table entries. The equivalenced variables are entered as a list of equivalenced groups (see Section 7).

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

12.1 Preprocessing equivalence groups

Before any space is allocated, the offsets of the equivalenced variables with respect to the leader of the group (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 conflicts 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 described in [Gri71].

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 was 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.

12.2 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, INITIALADDRESS, is obtained. It Is zero if the switch VARCOMMON is set because a static pseudo-level is reserved exclusively for each common area. If the switch is off and no space has been allocated for that area In any previously compiled program unit, the initial address is the current value of DISPL_GENCOMMON at level 2 for any common area except the blank common whose initial address is 0. If space has already been allocated for the common area, the initial address is the address where the area was previously allocated, stored in fields LEVEL and STADDR of the common name table entry.

If a common variable is also equivalenced, the variables In the same equivalence group are allocated using procedure ALLOC COMMON_AND_EQUIV called from procedure CHECK_EXTENSION, which also checks for invalid extensions to the left of a common area due to the equivalencing, After space is allocated for all the common variables of an area, extensions to the right of the common area are checked. See Section 7.6, The Common Table regarding how the initial length of a common area is determined.

12.3 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 variables not declared as common or equivalence is allocated.

13. P-Code generating routines

Almost all code that is written in the P-Code file Is generated by one of the P-Code generating routines, There are a few cases in which P-Code is written directly to the file by a main routine using WRITELN (P-Code, . . .

The main P-Code generating routine is GEN. This works for most instructions. There are four arguments: opcode, P-Code operand type, and two integers. Where not all of these are necessary, the superfluous ones are ignored. The P-Code operand type is of type GENTYPES, which is represented as the single-character P-code type doubled (see below). When no type is required, the type ZZDUMMY is passed. This is to make it clearer when reading the routine that calls GEN that no type is required, and also acts as a check to ensure that a type is passed whenever it is required. This check is performed by procedure PTYPE, which converts a variable of type GENTYPES to the actual string that is printed in the P-Code file.

```
GENTYPES = (AA,BB,CC,DD,HH,II,JJ,MM,NN,PP,QQ,RR,SS,XX,ZZDUMMY);
```

```
AA = address
BB = boolean
cc = character
DD = double word integer
HH = half-word integer
II = Single word integer
JJ = index
MM = multiple-unit arrays or records
NN = the ni I pointer
PP = procedure
RR = single word real
ss = the ordinal number for the element of a set
xx = double word real
```

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

GEN_LOADNUM -- the constant is to be taken directly from the FORTRAN statement kept in LEXSTRING. The pointer to the lexeme is passed

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

GEN_LOADINT, GEN_LOADREAL, GEN_LOADBOOL, GEN_LOADCHAR -- constant Is passed in integer, real, boolean and character forms respectively

Other P-Code generating routines are:

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

GEN_LABEL -- prints a P-Code label definition, e.g. "L15LAB"

GEN_DEF -- prints a P-Code constant definition, e.g."L16 DEF 20"

GEN_CMN -- generates a CMN instruction (not yet implemented)

GENCSP, GENMST, GENCUP, GENSST, GENENT -- generates the given instruction

GENSEG_CODE -- generates the dummy blocks (see Section 10)

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

PRINT_LABEL -- prints a P-Code label, e.g. "L15"

PRINT_NAME -- prints the name of a program unit in P-Code form, e.g. 'PEPE0003'. The maximum length 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.

14. Temporary storage management

Temporary locations are used in cases like storing loop variables and handling complex numbers. In order to be able to re-use these locations, two temporary storage management routines were written, which allocate and keep track of temporary locations of different sizes. All temporary locations used are at level 3 - the level for all variables except common and dummy variables.

FUNCTION GETTEMP (SIZENEEDED: INTEGER): INTEGER:

finds an unused temporary storage location and returns its address.

PROCEOURE RELTEMP (LOCATION: INTEGER):

indicates that the address indicated is no longer needed and may be used 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 nodes. The list is lengthened as more and more temporary locations are demanded in the course of compilation. The order of each node in the-list is not significant. The structure of each node is:

```
TEMPLOCNOOE = RECORD
                LOC,
                SIZE: INTEGER:
                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. If there is none, it claims a new one by incrementing DISPLACEMENT by 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.

15. Loading and storing variables

15.1 The procedures

The procedures used to generate code to load and store non-complex variables are LOADVAR, LOADVARADDR, LOAD ARRAY ELEMENT, and STOREVAR. (See the section on complex numbers for complex variables.) To load the value of a variable, LOADVAR is called. To store a value in a variable, LOADVARADDR is called, then the value is loaded (usually by ARITH) and then STOREVAR Is called.

There are three types of variables: simple variables, simple variables passed as parameters, and array elements. For the last two, 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 LOADVARADDR.

LOADVARADDR is passed a pointer to the symbol table for the variable In question. If the variable is a dummy variable, It loads Its address. If the variable is an array, it loads its address, if not already loaded (it may have just been loaded if the array Is also a dummy parameter), and then calls LOAD ARRAY ELEMENT, which reads the subscripts and generates code to calculate the offset. If the-variable Is either a dummy variable or an array, LOADVARADDR returns TRUE. Otherwise, it returns FALSE. LOAD ARRAY ELEMENT calculates the address is the following way: It checks to see if there is a left parenthesis, and then goes to the next lexeme. It then calls ARITH, which loads the integer expression corresponding to the first subscript on the stack. It then goes past the next comma, if any. For the second dimension, it generates code to multiply the value by the upper bound of the second dimension minus one before adding to the first subscript. For the third, it generates code to multiply the offset value by the upper bounds of both the first and second dimensions. When it encounters a right parenthesis instead of a comma, it returns.

For an array A of dimensions (X,Y,Z), then, the offset for A(a,b,c) would be

a + X*(b-1) + X*Y*(c-1).

15.2 *Example of Indirect* load *and store*

FORTRAN,

```
      SUBROUTINE X (I)
      DIMENSION J(3,4)
      J(2,3) = I
      RETURN
      END
```

P-code:

```
      SST P X0000031 5 4 0 0 4
X0000031 ENT P 5 L1 X0000031 1 1 1
      LDA 3 504          ;load address of array J
      LDC I 2
      DEC I,1
      LDC I 3
      DEC I,1
      LDC I 3
      MPI
      ADI
      IXA 4              ;up to here, load address of J(2,3)
      LOO A,5,8          ; load address stored at address of I
      IND I,0            ;load content of address just loaded
      STO I              ;store value at address 2nd on stack
      RET P
L1    DEF
```

16. Expression evaluation

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

Expression evalutation procedures are divided into three kinds: logical expression procedures, arithmetic expression procedures, and complex expression procedures. Logical expressions are expressions connected by logical operators, such as **".AND.".** They always include arithmetic expressions, which are constants or **varibles** or other arithmetic expressions connected by arithmetic operators.

ARITH, the expression evaluaator, expects the global lexeme pointer LXC to be pointing to the beginning of the expression when it is called, and leaves **it** pointing to the lexeme after the expression. It returns the **datatype** that will be left on the top of the stack when the expression is evaluated.

16.1 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    ari th expr
r e l  o p e r a t o r = ".LE." | ".LT." | ".GE." | ".GT." | ".NE." | ".EQ." |
                  " <"|" > " | " ="
arith expr = term laddop term)
term = {addop} factor {mul top factor)
factor = lpr i mary} {"**" pr i maryl
addop = "+" | "-"
mul top = "*"|"/"
primary = "(" ari th expr ")" | constant | complex constant | logical constant
          | variable | function call | array element
complex constant = "(" arith expr "," arith expr ")"
logical constant = ".TRUE."|".FALSE."
```

16.2 Processing identifiers

When **ARITH** encounters an identifier, it must determlne whether It is a variable, a call to **a** standard function, or a call to a user function,

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.

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,EXTENTRY,EXTFUNC,STMTFUNC,
            INTRINSTDEXT);
```

This is the way **ARITH** handles symbols:

1. Look it up in symbol table (this means that if it is not already there, it is entered, with, among other things, S FUNCSUCR set to NOTFUNC); If it has appeared in this program unit before, then S FUNCSUBR will already contain the Information about what kind of symbol it is;

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

3. else if we already know it is a standard function then call STANDAROFUNC

4. else if next lexeme is not an left parenthesis or it has been dimensioned, then it must be a simple variable or array element; call LOADVAR (see Section 16, Loading and Storing Variables).

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

6. 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

16.3 Example

FORTRAN: IF (3.2*I.EQ.5.1**3) . . .

Pcode:

```
        LDC R 3.2
        LOO I 3 500          ;load value of variable I
        FLT                  ;float value of I
        MPR
        LDC R 5.1
        LDC I 3
        CUP EXPON            ;call exponentiation function
        EQU R
```

17. Complex numbers

17.1 The complex stack

Unfortunately, complex numbers can't be handled directly by P-Code. Therefore, it is necessary to simulate the P-machine stack using a stack called CSTACK. It contains the addresses of all the complex numbers in the expression being evaluated. These addresses will be the addresses of either complex variables, complex constants (stored in temporary locations), or results of previous operations on the top of the complex stack (also stored In temporary locations).

It also contains information on whether the address location specified is a temporary location or the address of a regular complex variable. This is needed so that the temporary location can be released and reused after it is no longer needed.

The structure of CSTACK is as follows:

```
TYPE CSREC = RECORD
     ADDR: INTEGER;
     STORED-I N_TEMP: BOOLEAN
     END:

VAR CSTACK: ARRAY [1..MAXCSTACK] OF CSREC:
     CSTACKPTR: 0.. MAXCSTACK:
```

The P-machine stack is used merely to perform a single operation on the top or top two elements of the complex stack. Thus after every operation, it is empty.

The address of the final result is stored in the global variable CRESULTLOC. Thus, after any call to ARITH, the top of stack type should be checked. If it is of type complex, then it needs to be copied from CRESULTLOC, The real part of the number will be stored in CRESULTLOC and the imaginary part in CRESULTLOC + GETSIZE(TOPOFSTACKTYPE)/2.

The two operations on the complex stack are:

```
PROCEDURE PUSHCSTACK (ADDR: INTEGER: STORED-1 N_TEMP: BOOLEAN) ;

PROCEDURE POPCSTACK (VAR ADDR: INTEGER; VAR STORED_IN_TEMP: BOOLEAN);
```

17.2 Putting complex numbers on top of CSTACK

A complex number can be one of four types: a constant, a variable, a function result, or an array. In addition, it may have been passed as a parameter.

Procedure PRIMARY of procedure SIMPLE EXPRESSION checks to see if the current lexeme is a left parenthesis. If it is, it calls SIMPLE EXPRESSION recursively to evaluate the expression within parentheses. if, after doing that, it encounters a comma, then It knows it has found a complex constant. it gets a temporary iocatation using function GETTEMP, stores the real part of the expression into that location, calls SIMPLE EXPRESSION to get the imaginary part, and then copies that into the second half of the temporary location, It then pushes that address onto CSTACK.

If the current lexeme is a complex variable, then the address of that variable is loaded onto CSTACK. (This is done in procedure PROCESSID.)

If the current lexeme is an array or a parameter, then the final address is not known at compile time, For this reason, it is necessary to get another temporary location and generate code to copy the number there. That address is pushed onto the stack. This is all done In procedure LOADCOMPLEX,

If the current lexeme is a function call, then USERFUNC is called. The function result will be stored in the address pointed to by CRESULTLOC, which is pushed onto the stack.

17.3 Operations on complex numbers

Operations on complex numbers are defined as follows [Org66]:

If Z1= Al +i*B1 And Z2= A2 +i*B2, then

```
Z1 + Z2 = (A1 + A2) + i*(B1 + B2)

Z1 - Z2 = (A1 - A2) + i*(B1 - B2)

Z1 * Z2 = (A1*A2 - B1*B2) + i*(A1*B2 + B1*A2)

Z2 / Z1 = ((A1*A2 + B1*B2) / (A1**2 t B1**2))
      t i*( (A1*B2 - B1*A2) / (A1**2 t B1**2))
```

The procedures used for evaluating complex numbers are: CADDSUB, CMULT, CDIV, CNEG (unary minus), and CEXP (exponentiation). They all use the primative procedure COP, which generates code to load two variables on the P-stack, do a simple operation on them, and store the result in a third location.

17.4 Addition of two complex numbers

For an ADD operation the sequence of events would be:

0. do two POPCSTACKs to get the locations of the two numbers on the top of the complex stack

1. generate code to load the real part of the complex number second from the top of the complex stack onto the P-stack

2. generate code to load the real part of the complex number from the top of the complex stack onto the P-stack

3. get a temporary location to store the result

4. generate code to add the two numbers and store the result in the temporary location

6. repeat 1, 2, and 4 for the imaginary part

6. push the address of the result onto the complex stack

7. if either of the two complex numbers were in a temporary location, release the temporary location

Subtract, multiply, and divide are analogous.

17.5 Example of complex addition expression

FORTRAN: X = (3.,1.) t Y - (4.,2.)

P-code:

```
LDC R 3.0
STR R 3 516   ;store real part of (3.,1.) in temporary ioc.
LDC R 1.0
STR R 3 520   ;store imag. part of (3.,1.) in temporary loc.
LOO R 3 516
LOD R 3 508   ; load real part of Y
ADR           ;add real parts
STR R 3 524   ;store sum of real parts in temporary loc.
LOO R 3 520
LOD R 3 512   ;load imaginary part of Y
ADR           ;add imaginary parts
STR R 3 528   ;store sum of imaginary parts in temporary loc.
LOC R 4.0
STR R 3 516   ;store real part of (4.,2.) in temporary loc.
LDC R 2.0
STR R 3 520   ;store imag. part of (4.,2.) in temporary loc.
LOD R 3 524
LOO R 3 516
SBR
STR R 3 532   ;store difference of real parts in temp. loc.
LOO R 3 528
LOD R 3 520
SBR
STR R 3 536   ;store difference of imag. parts in temp. loc.
LOO R 3 532
STR R 3 500   ;store at real part of X
LOO R 3 536
STR R 3 504   ;store at imaginary part of X
```

17.6 Exponentiation

· Integer exponentiation is done as follows:

0. The exponent, being an integer, has been left on the P-stack. Generate code to store it in a temporary location

1. Get a temporary location for the loop variable; generate code to store 1 into it

2. pop the top of the complex stack to get the address of the base

3. get a temporary location for the accumulator and generate code to copy the base to that location for the first multiplication

4. push onto the complex stack the address of the base and the accumulator

6. generate a label to indicate the beginning of the loop

6. call the complex multiplication procedure; this will get a temporary location for the result, pop the addresses of the operand from the complex stack, generate code to do the multiplication and store the result in that location, and push the address of the result onto the complex stack

7. generate code to increment and test the loop variable and jump out if done

8. generate code to copy the result into the accumulator location

9. generate code to jump back to the label in 6

10. **release** all temporary locations

17.7 Example of complex exponentiation

FORTRAN: X =(5.,6.)**3

P-code:

```
        LOC  R 5.0
        STR  R 3 516    ;store real part of (5.,6.) in temp. loc.
        LOC  R 6.0
        STR  R 3 520    ;store imag. part of (5.,6.) in temp. loc.
        LDC  I 3
        STR  I 3 540    ;store 3 in temporary location
        LDC  I 1
        STR  I 3 544    ;store initial value of counter in temp. loc.
        LOD R 3 516
        STR R 3 524     ;store real part in temp. loc. to accumulate
                         result
        LOD R 3 520
        STR R 3 528     ;store imag. part in temp. loc. to accumulate
                         result
      L2 LAB
        LOD R 3 524
        LOD R 3 516
        MPR
        LOD R 3 528
        LOD R 3 520
        MPR
        SBR             ;compute real part of current multiplication
        STR R 3 548     ;store computed real part in temp. loc.
        LOD R 3 524
        LOD R 3 520
        MPR
        LOD R 3 528
        LOD R 3 516
        MPR
        ADR             ;compute imag. part of current multiplication
        STR R 3 552     ;store computed imag. part in temp. loc.
        LOD I 3 540
        LOD I 3 544
        LES I
        FJP L3          ;test loop termination
        LOD I 3 544
        INC I 3
        STR I 3 544     ;increment counter
        LOD R 3 548
        STR R 3 524     ;store real part in location to accumulate
                         result
        LOD R 3 552
        STR R 3 528     ;store imag. part in location to accumulate
                         result
        UJP L2          ; jump back for next multiplication
      L3 LAB
        LOD R 3 548
        STR R 3 500     ;store at real part of X
        LOD R 3 552
        STR R 3 504     ;store at imaginary part of X
```

18. The assignment statement

The assignment statement works as follows:

It first looks up the symbol in the symbol table and calls LOADVARADDR to load the address on the stack, if necessary (see Section 15). It sets the global lexeme pointer, LXC, to point to the lexeme after the equal sign. If the variable is a logical variable, it calls LOGEXPR. Otherwise it calls ARITH (see Section 16).

If the expression contains a string, then it calls the procedure STORESTRING (described below). Otherwise, if the expression is of type real and the variable of type integer or vice versa, then the appropriate P-Code instruction is generated to convert the expression. Any conversion between different sizes of integers and reals that Is necessary is handled automatically by SOPA. Any other mismatch between expression and variable types generate an error message.

If the expression is of type complex, then STORECOMPLEX is called (see Section 17). Otherwise, STOREVAR is called to generate code to store the variable (see Section 16).

STORESTRING is used to store a string into any kind of variable. It first checks to make sure that the expression consists of exactly one string constant. It then generates code to load the string, character by character, into the address Indicated. For simple variables, this is straightforward. The character is loaded onto the stack, and then code is generated to store it in the next quarter word of the address. For an array element or parameter, however, code must be generated to load the address for each character and increment It by the right amount. To do this, the address that was put on the stack by LOADVARADDR, which is still on top of the stack (the string has not been put on the stack yet) is stored in a temporary location. The address Is loaded from this location for each character and incremented using an INC Instruction.

19. 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 to TRUE whenever the compiler is processing a subprogram.

All the parameters of a function or a subroutine are passed by reference, thus the space that has to be allocated for them Is always 4 quarter words (the space required for an address).

Whenever a variable is processed 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 is 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 function and subroutines is kept in the external table instead.

19.1 Initialization of *a Segment Block*

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

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

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

3. In the common table, it sets the field 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.

4. it sets to FALSE the global variables AFTER STORAGE ALLOCATION, which indicates if the storage allocation of the variables de&red in the program unit has occurred, and HAS_RETURN, which indicates if a RETURN statement for the program unit has- been encountered.

5. It resets CURR_PCODE LABEL, the P-code label counter, and DUMMY_DISPLACEMENT, used to allocate space for the dummy arguments, COMMONSIZ, the variable in charge of the CSIZ option, is also reset to 0.

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

19.2 Subroutine Statement

After the call to INITBLOCK, routine SUBR STMT inserts the subprogram name In the symbol table with no type, level 6 and displacement 0. The symbol table is updated by a call to FEXTNAME. Then it calls procedure DUMMY_PROCESSING for the processing of the dummy arguments.

Procedure DUMMY_PROCESSING scans the parameters of a subroutine or a function, allocating space for them and Inserting their names, levels (always 6), addresses, and an indication that they are dummy arguments in the symbol table. It uses the global variable DUMMY_DISPLACEMENT for the allocation of space. DUMMY_DISPLACEMENT is initialized to the amount of space needed for the MST P-Code instruction (always 8 quarter words to allow for the return value of a function; see below). It is incremented by 4 for each parameter in the program unit.

7 9.3 Function Statement

Procedure FUNC_STMT initializes a new block, gets the type of the function if it is specifically indicated, gets its size modification if specified, inserts the function name in the symbol table Indicating its type, size and address (level 6, displacement 0), and processes its dummy arguments by calling procedure DUMMY_PROCESSING.

The return value of complex functions are not returned in displacement 0 at level 6 because 2 separate values have to be returned. Instead, space is allocated for it after the space reserved for the function parameters. The address of this space is the value returned by the function, and an indirect reference is needed, in the case of complex functions, in order to access the returned value of the function. For this reason, such functions are declared internally as being of type "address" for the SST and ENT P-Code instructions. (Function GETGENTYPE returns type AA for functions of types COMP8 and COMP 16).

19.4 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, since procedures must appear sequentially in P-code, even if they are nested.

19.5 Example

FORTRAN:

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

P-code:

```
      SST I X0000031 5 4 0 0 4
X0000031 ENT I 5 L1 X0000031 1 1 1
      LDC I 2        ; load constant 2
      LOD A 5 8      ; load address stored at address of I
      IND I 0        ; fetch content of this address
      MPI            ; compute 2*I
      STR I 5 0      ; store at address 0 for the return value
      RET I          ; generated due to the RETURN statement
      RET I          ; generated at end of all subprogram block
L1 DEF 12            ; stack frame of this subprogram is 12 words long
```

20. Subroutine and Function Calls

Dummy arguments of subroutines and functions are allocated addresses on 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 automatically by the P-Code machine, and is not visible in the P-Code program. In P-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.

There are three different treatments in the passing of addresses, depending on the type of actual parameter used in the call. If the parameter is a simple variable, an array name, or an array element, its address is put on the stack. If it is a string constant, it is loaded in the string constant area of the stack computer [giw77] and its address is then put on the stack. If it is an expression that is not just a single variable, code to evaluate the expression and put the resulting value on the stack is generated, followed by code to store this value in a temporary location and put the address of this location on the stack.

20.1 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 P-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.

Procedure PROCESS ARGUMENTS scans the arguments of a call. It differentiates four kinds of arguments: identifiers, array elements, string constants and expressions. For identifiers (simple variables and array names) and array elements, its address is loaded in the stack and a PAR P-Code instruction is generated. To recognize an array element, the boolean function IS_ARR_ELEMENT is used. For string constants, the string is stored in the string constant area of the machine with an LCA P-Code instruction, which leaves the address of the string on top of the stack, and then a PAR instruction is generated. Expressions are processed by the procedure ARGEXPRESSION, which works as follows:

It first generates code for the evaluation of the expression. Then it allocates space for the result of the expression, except for a complex expression whose result is stored In the location indicated by the global variable CRESULTLOC. It stores the result of the expression into the space just allocated, except for complex expressions that are already stored in memory. The procedure terminates by loading the address, where the result of the expression is stored, into the top of the stack and generating a PAR instruction to indicate that the address on top of the stack is a parameter.

20.2 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 FSYMBOL inserts the default FORTRAN type instead of NONE for the subroutine name.

Procedure USERFUNC is called to do the processing of the arguments and the generation of code for the call.

20.3 Example of a *function* call

FORTRAN:

```
COMPLEX*16 X
I = F(J,2*3,X)
```

Pcode:

```
MST 5 12 12        ;signal start of function call
LDA 3 524          ;load address of variable J
PAR A              ;first parameter
LDC I 2
LDC I 3
MPI
STR I 3 528        ;store value of expression in temporary
LDA 3 528          ;  location 528 and load this address
PAR A              ;second parameter
LDA 3 504          ;load address of variable x
PAR A              ;third parameter
CUP R 7 F0000031   ;end of function call code
TRC                ;convert returned value to integer
STR I 3 520        ;store at address of variable I
```

20.4 Standard Function Calls

Standard function calls are implemented in three ways:

1. 4 direct call to an equivalent P-code standard function (CSP).

2. In-line code.

3. A call to a function in the FORTRAN run-time package (CUP).

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

DESCRIPTION	NAME	ARGS	RESULT	PCOOE
absolute value	ABS	real	real	ABR
	IAB	int	int	ABI
	DABS	doubl	doubl	ABR
(mod)	CABS	complx	real	inline
truncation	AINT	real	real	TRC,FLT
	INT	real	int	TRC
	IDINT	doubl	int	TRC
mod	AMOD	real	real	inline

	MOD	int	int	MOO
	OMDO	doubl	doubl	inline
max	AMAX0	int	real	CUP MAXIN,FLT
	AMAX1	real	real	CUP MAXRE
	MAX0	int	int	CUP MAXIN
	MAX1	real	int	FLT,CUP MAXIN
	DMAX1	doubl	doubl	CUP MAXOB
min	AMIN0	int	real	CUP MININ,FLT
	AMIN1	real	real	CUP MINRE
	MIN0	int	int	CUP MININ
	MIN1	real	int	FLT,CUP MININ
	DMIN1	doubl	doubl	CUP MINOB
int to real	FLOAT	int	real	FLT
real to int	IFIX	real	int	CUP IFIX
transfer sign	SIGN	real	real	CUP SIGN
	ISIGN	int	int	CUP ISIGN
	OSIGN	doubl	cloubl	CUP OSIGN
positive diff	DIM	real	real	CUP DIM
(0 if a1<a2)	IDIM	int	int	CUP IOIM
doubl to real	SNGL	doubl	real	CUP SINGL
complex to real	REAL	complx	real	inline
complex imag to real	AIMAG	complx	real	inline
real to doubl --	OBLE	real	double	CUP ODUBL
real to complx	CMPLX	real	complx	inline
conjugate.	CDNJG	complx	complx	inline

exponential	EXP	real	real	CSP EXP
	OEXP	doubl	doubl	CSP EXP
	CEXP	complx	complx	not implemented
natural log	ALDG	real	real	CSP LOG
	OLDG	doubl	doubl	CSP LOG
	CLOG	complx	complx	not implemented
common log	ALOG10	real	real	not implemented
	OLDG10	doubl	doubl	not implemented
sin	SIN	real	real	CSP SIN
	OSIN	doubl	cloubl	CSP SIN
	CSIN	complx	complx	not implemented
cos	CDS	real	real	CSP CDS
	DCOS	doubl	doubl	CSP CDS
	CCOS	complx	complx	not implemented
tanh	TANH	real	real	not implemented
(IBM)	OTANH	doubl	doubl	not implemented
square root	SQRT	real	real	CSP SQT
	OSQRT	doubl	doubl	CSP SQT
	CSQRT	complx	complx	not implemented
arctan	ATAN	real	real	CSP TAN
	OTAN	doubl	doubl	CS? TAN
arctan (a1/a2)	ATAN2	real	real	OVR, CSP TAN
	DTAN2	doubl	doubl	OVR, CSP TAN

2 1. 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. The definition of this list is local to procedure STMT_FUNCTION:

```
DUMMY-L I ST = RECORD
                  PTR : PDINTSYMBDL;    (* points to its symbol table entry *)
                  S_FUNCSUBR: FUNCTYPE;
                  LEVEL,ADDRESS,
                  DIMENSION: INTEGER;
                  USED_LHS,
                  S-DUMMY: BOOLEAN;     (* original contents in symbol table *)
                  NEXT : ↑DUMMY_LIST;(* next in list *)
               END;
```

Procedure STMT_FUNCTION gets and inserts the name of the statement function in the symbol table with LEVEL field set to 6, ADDRESS field set to 0, USED_RHS set to false and USED_LHS set to true.

It processes the dummy arguments by calling procedure DUMMY_ARGUMENTS, which inserts them in the symbol table and remembers the old contents in the DUMMY_LIST records pointed to by HEAD_DUMMY. The dummy arguments are allocated addresses at level 6.

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, code is generated for the evaluation of the expression by calling procedures ARITH or LOGICALEXPR depending on the type declared for the statement function.

After that, code is generated to store the result of the expression in the space reserved for the statement function name at level 6. At the same time, code is generated to do the required type conversions.

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 table from the DUMMY_LIST records.

22. Do Loop

For each DO: code Is generated at 2 places: where the DO statement is recognized and at the end of the range of the DO. In the former, codee are generated for the initialization of the control variable of the loop, and a P-Code label is emitted to mark the beginning of the loop. In the latter, code is generated to increment the index variable by the appropriate 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 DO-list works as 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. CURRENT00 is a global variable that points to the record of the most recently opened do-loop at the top end of the stack.

The end of the range of a DO 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-list. At the end of a program unit, if there Is still any record on the DO-list, an error message is generated.

The DO-list is formed with the DOENTRY record of the form:

```
OOENTRY = PACKED RECORD
            CDNTRDLVAR : ↑SYMBOL;      (* POINTS SYMBOL TABLE ENTRY OF
                                          CONTROL VAR I ABLE *)
            STEPAMOUNT,
            UPPERAMOUNT : DIM;         (* STEP AND FINAL VALUES *)
            STMTLABEL,                 (* FORTRAN LABEL THAT ENDS THE
                                          THE RANGE OF THE LOOP *)
            PCDOELABEL : INTEGER:      (* PCDOE LABEL INSERTED WHERE
                                          THE DO-LOOP BEGINS *)
            PREV I DUS : ↑DOENTRY;     (* POINTS TO NODE OF PREVIOUS
                                          NESTED DO *)
            STEPKIND,
            UPPERKIND : BOOLEAN;       (* TRUE IF THE STEP OR UPPER AMOUNTS
                                          ARE GIVEN AS CONSTANTS,
                                          FALSE IF AS VARIABLES, *)
          END;
```

22.1 Do Loop Initialization

Procedure DOSTATEMENT scans and processes a DO statement. It creates an entry in the do-list, gets the FORTRAN label that terminates the range of the do-loop and inserts it in the entry just created, processes the control part of the do-loop by calling procedure DO_CONTROL and generates a P-Code label indicating the beginning of the do-loop.

In procedure DO_CONTROL, the control variable is located or inserted in the symbol table. Code is generated for the computation of its initial value and storage In the variable% memory location. The values or addresses of the final and increment values are saved in the most recently created DOENTRY record.

The initial value can be an integer expression, but the increment amount and the final

value must be an integer constant or integer variable. The default value of the increment amount is 1 if none is specified.

22.2 Do Loop Termination

Procedure CLOSED0 generates cocie for the termination of a do-loop. It is called by procedure BLOCK each time a FORTRAN label is found in the source code, in order to check if the label just found corresponds to the FORTRAN label that terminates the range of a do-loop, stored in the most recently created entry of the DO-list. 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 is so, code is also generated for its termination. This is repeated until the label in LABNO is not the end of the range of the current DO record.

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, GOT0 and arithmetic IF.

The generation of code for the termination of the loop is done in procedure GENCODE_FOR_DO.

. 22.3 DO loop example

FORTRAN:

```
      do 10 i=3,5,2          ; DO statemment
         ...
         code
         ....
10    continue               ; End of the range of the loop
```

P-Code:

```
      LDC I 3
      STRI,3,500             ; Store initial value of control var
     L2 LAB                  ; P-Code label to mark beginning of loop
        ...
  code for statements in
  the range of the do-loop
        ...
      LOOI,3,500             ; Load value of control variable
      INC I,2                ; Increment it
      STRI,3,500             ; Save i t
      LOOI,3,500             ; Reload it
      LDC 1 5                ; Load final value
      GRT I                  ; Compare them
      FJP L2                 ; Jump back if stillsmaller
```

23. GOT0 statements 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 P-code label is allocated for 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 LHS and RHS 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 P-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 GOT0 statements are processed as follows:

23.1 unconditional GOTO:

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

23.2 computed GOTO:

This compiles into the XJP instruction, which corresponds to the CASE statement of PASCAL. First, code to load the branch variable are generated by calling procedure LOADVAR, 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. In it, the UJP's for the list of statement labels are made, The form of the P-Code generated Is as follows:

FORTRAN statement: GOT0 (10,20,30),I

```
P-Code:          LOO I,1,500
                 XJP L40
               L40 OEF 0
               L41 OEF 2
               L42 LAB
                 UJP L11
                 UJP L22
                 UJP L33
               L43 LAB
                 call to exec error routines

Correspondences: L11- 10
                 L22 - 20
                 L33 - 30
                 500 - address of I
```

23.3 assigned GOTO:

Because P-Code labels referenced in P-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 P-Code. The first is to use the XJP instruction, which is like transforming the assigned GOT0 statement into the corresponding computed GOTO. The second method, which is the one used, does not use XJP, and generates denser P-Code. The label variable Is multiply loaded (by call of LOADVAR as in above) and its value compared one by one with each statement label in the list until equality is found. Then the corresponding jump is made. An example of the code generated is:

FORTRAN statement: GOT0 J,(10,20,30)

P-Code:

```
LOO I,520
LDC 10
NEQ
FJP L11        ; if I=10, jump to L11
LOO 1,520
LDC 20
NEQ
FJP L22        ; if I=20, jump to L22
LOD 1,520
LDC 30
NEQ
FJP L33        : if I=30, jump to L33
call to exec error routines
```

Correspondences:

```
L11 - 10
L22 - 20
L33 - 30
520 - address of J
```

24. The arithmetic IF and logical IF Statements

24.1 *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 P-code that evaluate the IF condition and put 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 variable 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 P-Code label which will be generated at 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 compiled, the P-Code label 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 lexemes for the statement start at position 1.

24.2 arithmetic IF

The arithmetic expression in the first part of this IF statement is processed by calling procedure ARITH, which will generate the P-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 3-way branch, two tests have to be made of the **value on** top of **the** stack. Since the address disappears after the comparison, 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 P-Code generated is:

FORTRAN statement: IF (...)10,20,30

P-Code:

```
. . .
(Code to evaluate expression and
 put result on top of stack)
. . .
STR I,1,504  ;store result on top of stack
LDC I,0
LOD I,1,504 ;reload i  t
GRT
FJP L11      ;if less than 0, jump to L11
LDC I,0      ;load 0
LOD I,1,504  ;reload expression
NEQ
FJP L22      ;if equal to 0, jump to L22
UJP L33      ;jump to L33
```

```
Correspondences: L11 - 10
                 L22 - 20
                 L33 - 30
                 504 - temporary location selected
```

(In this example, the arithmetic expression is assumed to be of type integer.)

25. The PRINT statement

The PRINT statement was intially written for use in debugging before the runtime I/O was working. It currently keeps track of whether files are open or not using its own procedure, OPENFILE. When the run-time is eventually linked in as an external, separately compiled procedure, it will use those file opening routines.

Procedure OPENFILE generates code to REWRITE (open for output) the file corresponding to the device number given if it is not already open. User file numbers -1, -2, -3, -4, -5, -6 correspond to PASCAL run-time files INPUT, OUTPUT, PRD, PRR, QRD, QRR. File number 0 corresponds to FILEO, file number 1 to FILE1, etc.

The normal way to print a numerical expression would be to load the file address, call ARITH to load the value on the stack, and the generate a call to WRI (write integer) or WRR (write real). The problem with this is that if the expression contains complex numbers, it is necessary to do at least one STORE during the expression evaluation. After a STORE, the stack must be empty, which it wouldn't be if you had already loaded the file address. therefore the sequence of events is:

If the next lexeme is a string, generate a LCA followed by a CSP WRS (write string); otherwise:

1) Call ARITH.

2) If the expression was complex, load the file address; then load first the real part and then the imaginary part from RESULTLOC (see Section 17) generating a call to WRR for each element.

3) For regular reals and integers, first store the value that was left on the stack In a temporary location, then load the file address, then load the value, and finally generate a call to WRR or WRI.

25.1 ***Example***

FORTRAN: PRINT 'X=',5:1,'Y=',(3.,2.)

P-code:

```
LDA 1 13        ;load address of file OUTPUT on the stack
CSP SIO
LCA 'X='
LDC I 2
LDC 1 2
CSP WRS
CSP EIO         ;write 'X='
LDC I 5
STR I 3 500     ;store 5 at temporary location 500
LDA 1 13
CSP SIO
LOO I 3 500
LDC I 1
CSP WRI
CSP EIO         ;write 5 in field of length 1
LDA 1 13
CSP SIO
LCA 'Y='
LDC I 2
LDC I 2
CSP WRS
CSP EIO         ;write 'Y='
LOC R 3.0
STR R 3 504
LDC R 2.0
STR R 3 508
LDA 1 13        ;load 3.0 and 2.0 from locations 504 and 508
CSP SIO
LOO R 3 504
LDC I 14
CSP WRR         ;write 3.0
LOO R 3 508
LDC I 14
CSP WRR         ;write 2.0
CSP EIO
CSP SIO
CSP WLN         ;end of line
CSP EIO
```

26. FORMAT Statement Processing

FORMAT statements are processed In two parts. 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= RECORD
              PTRFMTSTR : ↑FORMTSTR;(* POINTER TO THE FORMAT STRING LIST *)
              NEXT : ↑FORMTLIST;
              ADDRESS,
              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 : FDRMATSTRING;                (* FORMAT STRING *)
              NEXT : ↑FORMTSTR;
           END:
```

The purpose of this second list is to save space because it Is not necessary to acquire much more space than the maximum length of a FORMAT specification can have. Only increments of MAXCHARINLCA units of storage need be allocated by the compiler. MAXCHARINLCA is the limit on the length of the literal allowed in the P-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.

26.1 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 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: 1) the address and level allotted to the FORMAT label **and** 2) 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.

26.2 Initialization of Formats

Procedure INIT FORMATS is used to generate code for the loading of the FORMAT string specifications-&to memory at execution time. This procedure is called by procedure VARINITIALIZATION which Is in charge of all the initialization of variables for the compiler. (See Section 9.2, Procedure VARINITIALIZATION).

For each FORMTLIST record, procedure INIT FORMATS generates **a series of** LCA-LDA-MOV Instructions according the length of the FORMTSTR list. In each sequence 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.**

27. Read and Write Statements

27. 1 Run-time I/O routines

FORTRAN allows lists, loops, etc. within the Read and Write statements which allow fairly arbitrary complexity of variable sequences. in order to manage this complexity, the implementation conventions use multiple calls to system routines listed below:

27.1.1 Initialzation of i/O routines

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

FILE1029

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 to output execution error messages is open. An error flag for the I/O run-time routines Is initialized.

27.1.2 Initialization of single I/O statement

One call to an initialization routine before executing **each** Read/Write statement is required b**efore any data** transmission call can be made.

REAO I026
WRIT1023

Parameters: integer device number and address of FORMAT s tr i ng.

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.

27.1.3 Data transmission

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

REAOV028
WR I TV025

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 until 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,

27.1.4 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.

```
REAOT027
WRI TT024
```

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 and is set to

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

27.1.5 Rewind

Lastly, a call to

```
REW I N030
```

parameter: file number

is generated at a REWIND 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. This enables the user to start at the beginning of the file again for the same operation on the file.

2 7.2 *Compiler Routines*

Procedure IO_STATEMENT scans and processes the input/output 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          ; i f unformatted
```

where LIST is **a** list of variables that may only include simple variable names, **array names** and array elements. DEVICE 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, the two **temporary locations** remembered in MAXPRINTARRAY and CONPRINTARRAY **are always**

obtained when an I/O statement is processed. These temporary locations contain the upper bound (number of elements in the array) and index respectively for the array. They are released at the end of processing of the i/O statement.

Procedure IO_STATEMENT gets the device number and the FORMAT specification (either a FORMAT statement label or an array name), and generates code to call the run-time routines for the initialization for the I/O of the current statement, code for data transmission of the variables (by calling procedure LIST_PROCESSING) and code to call the routine 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 a do-implied or a list of variables surrounded by parentheses is found inside the list being processed. Parameter IN_00_IMPLIED indicates if the list of variables being processed belongs to a do-implied or is just a list of variables surrounded by parentheses.

For each element of the list, LIST_PROCESSING takes some specific action. If it is a simple variable, array element or an array, procedure VARNAME is called, If It Is a do-implied list, (procedure CHECK_DO_IMPLIED detects that), proceciure 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 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 P-Code is generated. This loop is preceded by, in their order, code to compute the number of elements of the array and store it in MAXPRINTARRAY, code to initialize CONPRINTARRAY, the indexing location, to 0 and a P-Code label to mark the beginning of the loop. Inside the loop are 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 are code which increment the index and test its value against that in MAXPRINTARRAY for loop termination conclition,

Procedure DO IMPLIED processes an implied do. First, it processes the control part of the do-loop using procedure DO CONTROL; then it generates code for the list of variables in the do-implied by calling procedure LIST PROCESSING with the parameter IN_DO_IMPLIED set to true; after this it generate&ode to close the do-loop using procedure CLOSEDO. Each do-implied has associated a dummy FORTRAN label (above 100000 to avoid any possible duplication with an existent FORTRAN label) that is used by the CLOSED0 routine. These dummy labels are not inserted in the label number table.

27.3 Code Generated

FORTRAN program:

```
INTEGER C(3,3),P(5)
..more code..
READ (4,8) (C, (P(I), I=N,M,1))
```

P-Code generated:

```
MST 2,8,8              ; Initiation.
LDC I 4
PAR I                  ; Load device number
```

```
    LOO A,3,564
    PAR A                   ; Load address of FORMAT string
    CUP P,5,READI026        ; Call to initialization routine

    LDC I 3                 ; I/O of array C
    LDC I 3
    MPI
    STR I,3,556             ; Compute size of array and store it in
    LDC I 0                 ;                MAXPR I NTARRAY
    STR I,3,560             ; Load initial value in CONPRINTARRAY
  L2 L A B                  ; Label that signals beginning of loop

    MST 2,12,12
    LDA 3,500
    LOD I,3,560
    IXA 4
    PAR A                   ; Load address of array element
    LDC I 4
    PAR I                   ; Load size of data value
    LDC I 0
    PAR I                   ; Load coded type
    CUP P,7,READV028        ; Call to data transmission routine

    LOD I,3,560             ; Load control variable
    INC I,1                 ; Increment it
    STR I,3,560             ; Save i t
    LOD I,3,560             ; Reload it
    LOD I,3,556             ; Load final value (from MAXPRINTARRAY)
    GEQ I                   ; Compare them
    FJP L2                  ; Jump back if not greater or equal

    LOD I,3,572             ; Do-implied with I/O of variable P
    STR I,3,568             ; Load initial value and save it in
  L3 LAB                    ;   control variable

    MST 2,12,12
    LDA 3,536
    LOO I,3,568
    DEC I,1
    IXA 4
    PAR A                   ; Load address
    LDC I 4
    PAR I                   ; Load size
    LDC I 1
    PAR I                   ; Load coded type
    CUP P, 7, READV028      ; Call to data transmision routine

    LOD I,3,568             ; Code to close the loop
    INC I,1
    STR I,3,568
    LOO I,3,568
    LOO I,3,576
    GRT I
    FJP L3

    MST 2,4,4               ; Termination of the I/O
    LDA 3,436
    PAR A                   ; Load address of indicator
    CUP P,3,READT027        ; Call to termination routine
```

28. The FORTRAN run-time package

The FORTRAN run-time routines are currently mostly I/O routines 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 FORTRAN run-time will eventually also include trigonometric functions, which will be written in S-I assembly language.

The I/O routines require the double precision facility in PASCAL to properly process the I/O of double precision variables in FORTRAN. Since this facility is not yet available, double precision I/O are processed only up to the accuracies allowed by single precision. The I/O of quarter-and half-word variables are completely handled.

The I/O routines are stored in P-Code form and copied to the end of the main P-Code file when necessary. They will eventualiy be stored in loader format along with the trigonometric functions, and linked to the main program by the linker,

28.1 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:

(A) error procedure - This outputs I/O execution error messages and sets error f lags.
  (1) procedure ERROR

(B) routines to handle the operations of the I/O buffer,
  (1) procedure CALLNEWOUTLINE
  (2) procedure NEWOUTLINE
  These write out the buffer as the next line in the output file.
  (3) procedure CALLNEWINLINE
  (4) procedure NEWINLINE
  These input the next line in the input file into the buffer.
  (5) procedure PUTCHAR - This puts the next output character to the I/O buffer,
  (6) procedure CETCHAR - This gets the next input character in the I/O buffer.

(C) procedures to process the FORMAT string.
  (1) procedure NEXTFIELD - When called, it will scan the format string 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.

(D) procedures for output conversions of data values.
  (1) procedure PRIFIELD - prints an integer in a I-formated field.
  (2) procedure PRFFIELD - prints a real in an F-formated field.
  (3) procedure PREFIELD - prints a real in an E-formated field.
  (4) procedure PRGFIELD - prints a real in an G-formated field.
  (5) procedure PRLFIELD - prints a boolean in an L-formated field.
  (6) procedure PRAFIELD - prints the contents of a variable in an A-formated field,

(E) procedures for formated input conversions of data values.
  (1) procedure REIFIELD - reads in an integer in an I-formated field.
  (2) procedure REEFGFIELD - reads in a real in an E-, F- or G- formated field, the effect being defined as identical.
  (3) procedure RELFIELO - reads in a boolean from an L-formated field. .

(4) procedure REAFIELD - reads in the characters in an A-formated field to a variable.

(F) procedures for unformated input conversions of data values.
(1) procedure UNFINTINPUT - scans and inputs an integer.
(2) procedure UNFREALINPUT - scans and inputs a real number.
(3) procedure UNFBOOLINPUT - scans and inputs a boolean.

(G) procedures called externally.
(1) procedure WRITINI (P-Code name is READ10261
(2) procedure WRITTRM (WRITI023)
(3) procedure WRITVAL (WRITV025)
(4) procedure READINI (READI026)
(5) procedure READTRM (READT027)
(6) procedure READVAL (READV028)
(7) procedure FILEINI (FILEI029)
(8) procedure REWIND (REWIN030)

In WRITVAL and READVAL, for formatted I/O, (C) NEXTFIELD is first called, and then the appropriate procedure In (D) or (E). For unformated I/O, in WRITVAL, the standard field widths is assigned and the appropriate procedure In (D)(1), (3) and (5) Is called. In READVAL, the appropriate procedure in (F) is called. Note that the procedures in (D), (E) or (F) treat the transmitted data value as double word size. WRITVAL will do the necessary shifting for smaller sized data values before calling (D). READVAL will do the necessary shifting after calling (E) or (F). 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.

28.2 Processing the FORMAT string

The entities allowed in a FORMAT string are: numbers, Hollerith string, literal string (enclosed in quotes), comma, slash, X, (,), 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. Same holds for GPCOUNT2 and 2nd level group. GPBEGIN1, GPBEGIN2 and GPBEGIN3 give the starting position of the current group of the corresponding level.

When scanning reaches the end of the FORMAT string and still has yet to look for the next I/O field, back-up has to occur to the beginning of the last 2nd level group. For this purpose, LASTGPPOS and LASTGPREP will hold te 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 until scanning reaches 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.

28.3 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 all taken to be the blank character.

28.4 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 string, 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 value 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 right values to be transmitted.

In the implementation, the data type

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

allows access to 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 by shifting the value prior to output conversion or shifting 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.

28.5 Output conversions of data values

All 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 all output conversions, variable IOBUFCURS always points to the left boundary of the output field. Another variable W1 indexes across the width of the field. The FOR loop is always used, and W1 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 11 digits) if < .34359738367, and by 10 ** 10 otherwise, to convert to an integer. 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 11 digits before the decimal, the less significant digits are made all 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-D-5)) OR (OUTREAL < 0) AND (0 c (W-D-6)) OR
   (S > (W-D-5)) OR (OUTREAL < 0) AND (S < (11-D-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 allowed.
   Note that this is different from the smallest representable
   real number, which has the lowest power but without the
   mantissa normalized,)

3. ELSE (a) get sign if negative
   (b) normalize OUTREAL to >= 0.1 and < 1.0, and
   accumulate the power in variable E

```
(c) I F ((S+D) >= 0) A N D ((S+D)<= 10)    (Here, 10 is
      largest number of significant digits stored in
      a word of memory)
    THEN OUTREAL := OUTREAL t 0.5 * 1 0 ** (-(S+D))
    (Do rounding.  (S+D) is the number of significant
     digi ts printed. )
(d) I F OUTREAL > 1.0 (increase to > 1.0 due to rounding)
    THEN BEGIN OUTREAL:=OUTREAL / 10;
               E := E t 1 END
(e) I F OUTREAL < .34359738367
    THEN CURTRUNC : = TRUNC(OUTREAL * (10 ** 11))
    ELSE CURTRUNC : = TRUNC(OUTREAL * (10 ** 10))
(f) output digits from CURTRUNC, the dicimal point being
    governed by S.
(g) E := E - S;
    print the exponent according to E.
```

28.6 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, variableIOBUFCURS 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 G- 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]) AN0
      (W1 is within boundary) DO
   BEGIN
      process this character
      Wl := 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 straigtforward structure:

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

```
                THEN OONE := 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 = 34369738367)

```
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 representible 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 t (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.

29. References

[NAJ75] K. Nori, U. Amman, K. Jensen, et al., "PASCAL P Compiler Implementation Notes", ETH Zurich, 1975.

[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] ANSI: "USA Standard FORTRAN", USA Standards Institute, USAS X3.9-1966, New York 1966.

[ANS71] 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, Vol. 11, No. 3, March 1976, (254 pages).

[BrW68] Gary Y. Breitbard and Gio Wiederhold, "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.

[FiZ78] Jim Finnel and Pol le T. Zel lweger, "The S-l Multi-processor", DSL Technical Note 142, Stanford University, June 1978.

[Gri713 David Gries, "Compi ler Construction for Digital Computers". John W i ley and Sons, 1971, pp. 304-312.

[JeW75] K. Jensen, and N. Wirth, "PASCAL User Manual and Report", Spr inger Ver lag, New York, 1975.

[kew78] Arthur Keller and Gio Wiederhold, "S-l Intermediate Loader Format", S-l project document LDI-8, 27Nov78.

[Org66] El liott I. Organick, "A FORTRAN IV Primer", Addison-Wesley, 1966, p.48.

[giw77] Erik J. Gilbert and David W. Wall, "P-Code Intermediate Assemb l y Language", S-l project document PAIL-3, 18JUL77.

[gwa78] Erik J. Gilbert and David W. Wall, "Specification for Run-time Suppor t for PASCAL", S-l project document PRUN-0, 20MAR78.

[wag78] David W. Wall and Erik J. Gilbert, "SOPAIPILLA Maintenance Manua l", S-l project document SOPADOPE-1, 23Mar78.

CWiB701 Gio Wiederhold and Gary Breitbard, "A Method for Increasing The Modularity of Large Systems", IEEE Computer, Vol. 3, no. 2, March-Apr i l 1970, page 30.

APPENDIX:

NOTES ON RUNNING PCFORT AT W-AI:

Compiling and running a Fortran program using PCFORO involves keeping track of a lot of files. So, it is better to use a DO file, namely FOR.DO[FOR,S1].

Due to line-length constraints, what the DO file does is: copy your file to the file X.FOR[FOR,S1], alias to that area, do the necessary things, and then alias back to your area to run the program. Therefore, the do program needs to know your PPN as well as the name of the source file: (The source file can be on any area.)

```
DO FOR [FOR, S1]

?f= HYDRO.FOR
?p= 1,PN
```

This DO file assumes that the Fortran program is to be executed on the S1. SOPA is run to translate the output P-code into S1 code. During execution on the S1 simulator, file OUTPUT will contains execution error messages, and file FILE01 will contain output written to device 1, FILE02 to device 2, etc.

Specific breakdown of FOR.DO:

FOR(1): copies source file to X.FOR, aliases to (FOR,S1)

FOR(2): X.FOR to X.PCO (PCFOR0)

FOR(3) : X.PCO t o X.LDI (SOPA)

FOR(4): al iases back to your area, runs X.LDI[FOR,S1](FSIM)

Various files are produced and reside in the [FOR,S1] area as the result of the two-stage compilation of the Fortran program, apart from those that come from executing the object program. They are:

X.FOR: copy of the original Fortran source program
X.LST: compilation l isting of the Fortran program (including error messages)
X.ERR: this lists only the compilation error messages of the For tran program
X.PCO: this contains the translated P-code of the Fortran program
X.PS1: SOPA listing of the P-code to S1 code translation
X.LDI: this contains the executable S1 code output by SOPA

If the user wishes to run the various phases of the compilation separately, he may construct separate 00 files without linking them together.