A PFORT Preprocessor

Peter Y. T. Chan

Department of Computer Science
University of Waterloo

Research Report CS-76-34

June 1976

A PFORT PREPROCESSOR

by

Peter Y.T. Chan

A thesis
presented to the University of Waterloo
in the partial fulfillment of the
requirements for the degree of
Master of Mathematics
in
the Department of Computer Science

Waterloo, Ontario, 1976

ABSTRACT

This thesis describes the design and implementation of a compiler which has portable Fortran (PFORT) as its target language. A phrase-structured language is defined which facilitates the writing of well-structured, easily-understood programs and a table-driven translator, whose target language is portable Fortran. The translator is implemented using a compiler-compiler.

Both the design of the source language and its implementation are done with portability objectives in mind. The source language constructs have been chosen in such a way that the translated Fortran output is efficient and highly readable. The language can be enhanced by simply adding or changing a few production rules and the corresponding semantic routines.

It is felt that a Fortran preprocessor, besides providing better control structures for the language (which are trivial to implement), should remove the difficulties of using a portable subset of Fortran. A number of problems arise during the implementation of the preprocessor. However, it is feasible and useful to have such a tool for portable programming.

ACKNOWLEDGEMENTS

This thesis is originated in the idea of designing a portable preprocessor by Michael A. Malcolm and Lawrence D. Rogers. Both of them have contributed to the project in many ways: by encouraging my interest in the subject of portability, by patiently monitoring and directing my research with countless helpful suggestions. One of the most important factors in the completion of this thesis has been Michael Malcolm's willingness to read and extensively annotate early drafts of the thesis.

TABLE OF CONTENTS

INTRODUCTION

1. Introduction

Portability is desirable for the easy exchange of programs between people or institutions and a decrease in the expenses of moving programs to a new machine. Portable programs are more economical because they do not need to be extensively revised by hand to fit a new computer system.

Automatic translators, decompilers, standardization of languages and machines, and portable compilers that can be bootstrapped onto a new machine using a macro processor are suggested methods for overcoming the great diversity of machines and programming languages. The most extensive and successful work in standardization has been in Fortran.

Fortran has many deficiencies. The control structures are poor. There is no statement grouping ability; the IF statement is primitive and there can be no ELSE part; the Fortran DO loop is no better, it restricts the user to going forward in an arithmetic progression with a minimum trip count of one. Programmers have to use a different built-in function name for each argument type. Mixed mode operations between integer and real or double precision operands are not allowed. There are many other restrictions (especially in ANSI Standard Fortran) which make programming in Fortran difficult and the source code hard to read.

A program written in Fortran is not necessarily machine independent. Every vendor (and many installations) writes its own compiler which does not compile the same language. Local modifications of Fortran have been made in many installations to remove some of the deficiencies.

On the other hand, Fortran has been the most widely used scientific programming language. One reason for this is the advanced development of Fortran compilers. Other reasons include the existence of subroutine libraries, the availability of Fortran on most computers, and the fact that Fortran is the most portable language currently available.

Extensive work has been done on standardizing Fortran to enhance the portability of programs written in Fortran. To aid in determining whether a Fortran program is really written in a portable subset of Standard Fortran, the PFORT verifier (see Ryder(1974)) has been written to check for syntax errors and inter-program communications, which includes improper matching of actual arguments with dummy subroutine arguments, recursion, and unsafe references. However, programming in the portable subset of Fortran involves discipline on the part of the programmer and a thorough knowledge of exactly what "Standard Fortran" is.

This thesis presents the design and implementation of a compiler which has portable Fortran (PFORT) as its target language. The proposed new language, called JOTS, is by no means meant to be a perfect programming language. Many "desirable" features have been omitted for various reasons. However, the language is still rich enough to facilitate the writing of well-structured, easily understood programs.

JOTS greatly reduces the burden in producing portable software. Readable Fortran output is produced so that it can be used as a basis for code maintenance by the user. This was suggested by Malcolm and Rogers (1974). However, another way to maintain a program developed with the aid of a Fortran preprocessor is to transport the preprocessor source code along with the preprocessor itself. The preprocessor has been implemented using the YACC compiler-compiler system (see Johnson (1975)) on the Honeywell 6060. The Eh language (see Braga (1976)) has been used for writing of the actions and support routines. The preprocessor is portable if it is implemented using a portable system. The descision for choosing Eh for implementing the JOTS translator has been influenced by the work done by Malcolm and Sager (1976).

In their project, they have designed and implemented a portable set of systems software on several machines including the Honeywell 6060. As the first part of the project, they have designed and implemented Eh which will be common to all machines. While a high-level language does not guarantee portability, the prime design criterion for Eh has been to insure that it will permit and encourage portable programming.

Sections 6.1 and 6.2 give a detailed decription of the implementation procedure. Since Eh is portable, the JOTS compiler is portable.

Detailed design objectives for JOTS and its preporcessor are outlined in chapter 3. The JOTS language description is in chapter 4. A number of problems arise in the implementation of such a preprocessor; these problems are discussed in chapter 5. Solutions to these problems and other implementation details are described in chapter 6.

2. Review of related work

Gales (1975) suggests techniques for structured programming in Fortran with no preprocessor. The technique is limited to those Fortran compilers which permit comments to be appended to a line of executable code. It involves the careful coding of Fortran using a rigid style. To do this more efficiently and to eliminate errors, the translation process can be automated.

Most Fortran preprocessors aim at adding "disciplined" control structures to the Fortran language. The control structures include statement groupings, alternative constructs, iteration constructs, general multiple level exits, internal procedures and macros. The extended Fortran control structures of twenty preprocessors are discussed in Meissner (1975).

Most Fortran preprocessors are implemented as translators which examine each statement of the source program to see if it is an extended statement (a statement valid in the preprocessor language but not in Fortran). If it is recognized as an extended statement, the translator generates the corresponding Fortran statements. If the statement is not recognized as an extended statement, the translator assumes it must be a Fortran statement and passes it through unaltered. Thus the translator does not restrict the use of

Fortran statements and compiler-dependent constructs are allowed.

The portability of programs developed with the aid of these preprocessors is somewhat enhanced by porting the preprocessors themselves, when that is possible.

Following is a discussion of the weaknesses in most preprocessors.

Syntax errors

Fortran syntax errors are not detected by the translator but by the local Fortran compiler which then prints a message in terms of the generated Fortran. In some cases this may be difficult to relate back to the offending line in the original source, especially if the implementation conceals the generated Fortran.

Readability of source program

Many preprocessors allow only fixed-form input as in Fortran. Numeric labels are used in the source by most preprocessors so that the source is sometimes hard to read. In most preprocessors, variable names are limited to a length of at most six characters. A few preprocessors (as well as many Fortran compilers) allow longer variable names but truncate them to six characters during the input phase.

Readability of object Fortran code

Most preprocessors consider the object Fortran code to be useless for code maintenance and debugging purposes. The generated Fortran code is unreadable. Comments are not copied to the object Fortran code.

Portability of source programs

Non-standard Fortran statements are allowed in all preprocessors (known at the time of this writing) so that the object Fortran code may not be portable. However, several preprocessors have been designed to be portable. It should be noted that even when a preprocessor is portable, programs developed using this preprocessor are not portable since some Fortran compilers may not support the non-standard Fortran constructs used in the programs processed by the preprocessor.

Implementation weaknesses

Most of the preprocessors have implementation weaknesses. For example, programmers must remember which ranges of numeric labels or what forms of variables are generated by the translator so that there will not be conflicts between user defined variables and translator generated variables. Most of these restrictions arise because the preprocessors are implemented as simple-minded translators (often macro processors) rather than complete compilers.

RELATED WORK

Ratfor (see Kernighan (1975)) is an exception to many (but not all) of the above observations. Since Ratfor has objectives similar to those proposed in this thesis, it is chosen to be discussed here.

RATFOR

The Ratfor language (Rational Fortran) is Fortran except for two aspects - the control flow structures and options to bypass some "irrational" restrictions of Fortran. The control flow aspect includes statement groupings, IF-ELSE statements, DO statements, BREAK for leaving a loop early, NEXT for beginning the next iteration, WHILE statements, and FOR statements. Ratfor also includes options for free-form input, translations of quoted strings into Standard Fortran Hollerith constants, and translations of comparison operators into corresponding Fortran operators (for example ">" to ".GT.").

Ratfor was developed using the UNIX YACC compiler-compiler. It is written in Ratfor and hence it is portable. However, Ratfor still has many weaknesses.

The grammar includes, in addition to rules for the Ratfor statements, a rule which assumes any unrecognizable line of code is a Fortran statement. Such unrecognizable statements are accepted and passed through. Ratfor allows free-form input but has some implementation weaknesses. For example, the continuation convention in Ratfor is that when a line ends with a slash '/' it is continued, since the slash is probably an arithmetic operator. But the Fortran DATA statement also ends with a slash. Thus one must terminate each DATA statement in a Ratfor program with a semicolon. Fortran syntax errors are not detected by Ratfor but by the local Fortran compiler. The object Fortran code generated by Ratfor is unreadable.

The design and implementation of the Fortran preprocessor described in this thesis aim at both the portability of the Fortran output and the portability of the preprocessor. The translator is implemented as a complete compiler so that deficiencies and restrictions of Fortran can be bypassed at the same time. The Fortran output is highly readable. The compiler also detects all syntax errors. A Fortran program that has been translated from a source program with no syntax errors is free from Fortran syntax errors.

3. Design objectives

With the motivation described in Chapter 1 in mind, the design objectives for a portable Fortran preprocessor are discussed in this chapter.

3.1 The source language

Well-chosen data types and control structures should be used subjected to the restriction that the translated Fortran output should be readable and portable.

Control Structures

Some means of statement grouping is needed. It is essential to have alternative constructs and iterative constructs. A small set of control structures which is rich enough to facilitate the writing of well-structured easily-understood programs and can be translated into readable Fortran code, should be chosen.

Since the Fortran DO-loop has a minium trip count of one, it is unwise to include a source language DO-loop that is to be translated directly to a Fortran DO-loop. Entry points in the middle of a subprogram unit often make the control flow hard to understand. The source language should force a program unit to have its only entry at the top and its only exit at the bottom.

Source program readability

Source input to the preprocessor should be free-form. The comment convention should allow comments and code to co-exist on the same line to allow remarks. The use of symbolic comparison operators (such as ">" and "<=") instead of Fortran operators (such as ".GT." and ".LE.") makes the code more readable. Long identifier names (more than 6 characters) should be allowed for readability and to encourage self-commenting code.

Keywords for the source language should be reserved to help the readability of the source programs. Many compiler-compilers (including YACC) require keywords to be reserved.

3.2 The Fortran output

In order that the Fortran output is portable, the ANSI Standard Fortran (1966) and the PFORT verifier should be used to determine the portable subset of Fortran that is to be used.

The Fortran code should be readable. Spacings between operators and operands help human readers observe the precedences of operations in an expression. Indentations help to understand the statement groupings and control flow of the program. The Fortran code is more readable if it is formatted to reveal the control structures of the source code as closely as possible. Neatly transcribed comments are essential for readability. Identifiers in the source should be mapped into unique and "readable" Fortran names.

A Fortran program produced by the translator from a source program with no detected errors should be free of syntax errors when it is processed by a Fortran compiler.

For an expression a*b*c, some Fortran compilers generate code such that a*b is computed first, while some optimizing Fortran compilers may generate code so that b*c is computed first. When floating point arithmetic is considered, operations such as multiplication and addition are not associative. Moreover, function calls often have side effects. Hence different results for such an expression may be produced when the same program is compiled and executed on different machines. In order to avoid this, expressions in the Fortran output should be fully-parenthesized to ensure a specific order of evaluations.

3.3 The translator

The translator should be portable. The translator should detect all syntax errors and some of the semantic errors in the source program. Readable error messages are useful. Compiler options such as symbol table dump and source program listing are also useful. Provisions should be made to allow a programmer to turn these options on or off.

With the above design objectives, it is clear that the preprocessor must be large and complicated. To program in the Fortran portable subset is often considered to be a difficult task. In addition, Fortran is not a very good language choice for implementing compilers; and the resulting execution modules tend to be considerably larger than those of a good systems implementation language. Hence, Fortran would be a poor choice for the implementation of the preprocessor. The advanced technology of compiler-compilers simplifies the implementation of a preprocessor and such preprocessors are easily modified. The translator should be implemented using a portable systems implementation language to achieve portability of the preprocessor.

4. Language Description

The grammar of JOTS, written in a Backus Normal Form, produces an LALR(1) language. Hence the language can be parsed from left to right with a local lookahead of at most one symbol. The compiler has been implemented using the compiler-compiler YACC (see Johnson (1975)).

4.1 Notation

In the following description of the grammar, the term "list" is normally used to denote one or more items separated by commas or semi-colons. For example :

```
<stmt list>     ::= <stmt>
                  | <stmt list> ; <stmt>
```

and

```
<identifier list>   ::= <identifier>
                      | <identifier list> , <identifier>
```

For practical reasons, the grammar is written with a substantial number of abbreviations. The following is a summary of them :

arith	arithmetic
arg	argument
auto	automatic
decl	declaration
desc	descriptor
err	error
expn	exponent
expr	expression
extrn	external
fcn	function
fmt	format
head	heading
init	initial
op	operator
opt	option
param	parameter
paren	parenthesis
stmt	statement
subpgm	subprogram
subr	subroutine
var	variable

<empty> is used to denote the null terminal symbol.

Unless otherwise specified in the particular section, all occurrences of the symbol T within a syntactic rule represents any of the following words:

integer
real
longreal
string
logical

For example, the grammar rule

```
<T var>                 ::= <T array designator>
```

corresponds to

```
<integer var>           ::= <integer array designator>
<real var>              ::= <real array designator>
<longreal var>          ::= <longreal array designator>
<string var>            ::= <string array designator>
<logical var>           ::= <logical array designator>
```

LANGUAGE DESCRIPTION

The description of the grammar is given as a set of productions, the non-terminal <program> is the goal symbol.

4.2 Vocabulary, lexical tokens and syntactic entities

4.2.1 Vocabulary

```
<character>      ::= <letter>
                   | <digit>
                   | <pseudo digit>
                   | <special symbol>
                   | <quote>

<letter>         ::= A | B | C | D | E | F | G | H | I
                   | J | K | L | M | N | O | P | Q | R
                   | S | T | U | V | W | X | Y | Z
                   | a | b | c | d | e | f | g | h | i
                   | j | k | l | m | n | o | p | q | r
                   | s | t | u | v | w | x | y | z

<digit>          ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<pseudo digit> ::= _

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

<quote>          ::= '
```

"␢" denotes the blank character.   The   lexicographical order of the characters is as follow:

b̸ ! " # ¢ % & ' ( ) * + , - . / 0 1 2 3 4 5 6 7 8 9

: ; < = > ? @ A B C D E F G H I J K L M N O P Q R S T U

V W X Y Z [ ] a b c d e f g h i j k l m n o p q r s t u

v w x y z ¦ ~

With the exception of characters in a string constant or comment, all lower case letters are converted into upper case during the input phase.

4.2.2 Lexical tokens

Special tokens

```
,   ;   :   .   (   )   [   ]   (/   /)   :=   +   -   *
/   **   $   %   >   <   >=   <=   ~=   =   !
```

"(/" is interchangeable with "[" and "/)" is interchangeable with "]".

Reserved words

ABS AND ARRAY ATAN BEGIN CALL CEILING COS DO ELSE END ENDFILE ERR EXIT EXP EXTERNAL FALSE FLOAT FLOOR FORMAT FUNCTION GOTO IF INTEGER LOG LOG10 LOGICAL LONG LONGREAL MAIN MAX MIN NOT OR PAGE PRINT READ REAL RECORD RETURN REWIND ROUND SHORT SIGN SIN SKIP SQRT STRING SUBROUTINE THEN TRUE TRUNCATE WHILE WRITE

All reserved words may be represented in either upper or lower case letters with no intervening blanks.

Identifier

```
<identifier>   ::= <letter>
                 | <identifier> <letter>
                 | <identifier> <digit>
```

```
                    | <identifier> <pseudo digit>

<identifier list>   ::= <identifier>
                    | <identifier list> , <identifier>
```

Identifiers are restricted to lengths of no more than 160 characters. Subprogram names are restricted to lengths of no more than six characters, and they may not include pseudo digits. They are referred to as <subpgm identifier> in later sections.

Examples of identifiers are:

```
machine_eps
a1
input_string
```

Integer, and real constants

```
<unsigned integer>  ::= <digit>
                    | <unsigned integer> <digit>

<signed integer>    ::= + <unsigned integer>
                    | - <unsigned integer>
                    | <unsigned integer>

<integer constant>  ::= <unsigned integer>

<real constant>     ::= <unscaled real>
                    | <unscaled real> <real expo>
```

```
                         ! <unsigned integer> <real expo>

<unscaled real>        ::= <unsigned integer> .

                         ! . <unsigned integer>

                         ! <unsigned integer> . <unsigned integer>

<real expo>      ::= E <signed integer>
```

Examples of real constants:

```
3.1415926
1E0
8.
```

String constant

```
<string constant>    ::= ' <string body> '

<string body>   ::= <string character>

                         ! <string body> <string character>

<string character>   ::= <character> ! ''
```

The length of a string is defined as the number of <string character>s in the body of the string. "''" is a string character representing a single quote.

Examples:

```
'John''s son'

'''length is 14'''
```

4.2.3 Syntactic entities

The following is a listing of all the syntactic entities with the section numbers.

Syntactic entity	Section number
<a var>	4.4.3
<a var list>	4.4.3
<adjustable dimension>	4.4.8
<alphabet>	4.2.1
<arith constant>	4.4.5
<arith op>	4.6.3
<arg>	4.5.4
<arg list>	4.5.4
<array decl>	4.4.3
<assignment stmt>	4.5.3
<auto array arg>	4.5.4
<auto indicator list>	4.5.4
<basic type>	4.4.2
<bound>	4.4.8
<bound list>	4.4.8
<built-in T fcn>	4.6.6
<call stmt>	4.5.4
<character>	4.2.1
<compound stmt>	4.5.2
<control head>	4.5.8
<decl>	4.4.1
<decl fmt>	4.4.9
<decl fmt list>	4.4.9
<decl list>	4.4.1
<declarations>	4.4.1
<digit>	4.2.1
<edit desc>	4.7
<euf opt>	4.5.8
<empty>	4.1.1
<endfile stmt>	4.5.9
<entry list>	4.3.2
<entry name>	4.3.2
<err opt>	4.5.8
<fcn>	4.3.2
<fcn decl>	4.4.6
<fcn head>	4.3.2
<field width>	4.7
<file unit>	4.5.8
<fmt decl>	4.4.9
<fmt group>	4.7
<fmt repeat list>	4.7
<fmt specifier>	4.5.8

<fractional desc>	4.7
<fractional digit>	4.7
<fractional field>	4.7
<goto stmt>	4.5.5
<identifier>	4.2.2
<idenifier list>	4.2.2
<if clause>	4.5.6
<if stmt>	4.5.6
<init group>	4.4.5
<init repeat list>	4.4.5
<init value>	4.4.5
<input control>	4.5.8
<input opt>	4.5.8
<input opt list>	4.5.8
<integer constant>	4.2.2
<integer expr>	4.6.3
<io array>	4.5.8
<io item>	4.5.8
<io item list>	4.5.8
<iteration stmt>	4.5.7
<label head>	4.5.1
<label identifier>	4.5.5
<letter>	4.2.1
<logical constant>	4.6.5
<logical element>	4.6.5
<logical expr>	4.6.5
<longreal expr>	4.6.3
<lower range>	4.4.4
<main program>	4.3.1
<many-arg list>	4.6.6
<many-arg T fcn>	4.6.6
<newline>	4.2.4
<one-arg fcn name>	4.6.6
<one-arg T fcn>	4.6.6
<output control>	4.5.8
<output data>	4.5.8
<param array decl>	4.4.7
<param bound list>	4.4.8
<param decl>	4.4.7
<param decl list>	4.4.7
<print stmt>	4.5.8
<program>	4.3
<program body>	4.3.3
<program unit>	4.3
<program unit list>	4.3
<pseudo digit>	4.2.1
<quote>	4.2.1
<range>	4.4.4
<range list>	4.4.4
<range specifier>	4.5.8
<range specifier list>	4.5.8
<range value>	4.4.4
<read stmt>	4.5.8

<real constant>	4.2.2
<real expr>	4.6.3
<relational op>	4.6.5
<repeat factor>	4.4.5
<returned expr>	4.3.2
<rewind stmt>	4.5.9
<s var>	4.4.3
<s var list>	4.3.3
<scale factor>	4.7
<signed integer>	4.2.2
<simple type>	4.4.2
<simple var decl>	4.4.3
<special symbol>	4.2.1
<stmt>	4.5.1
<stmt list>	4.5.1
<string body>	4.2.2
<string character>	4.2.2
<string constant>	4.2.2
<string element>	4.6.4
<string expr>	4.6.4
<string size>	4.4.2
<subpgm>	4.3.2
<subpgm decl>	4.4.6
<subpgm identifier>	4.2.2
<subpgm identifier list>	4.4.6
<subr>	4.3.2
<subr decl>	4.4.6
<subr head>	4.3.2
<subscript>	4.6.2
<subscript list>	4.6.2
<T array designator>	4.6.2
<T left part>	4.5.3
<unlabelled stmt>	4.5.1
<unscaled real>	4.2.2
<unsigned integer>	4.2.2
<upper range>	4.4.4
<write stmt>	4.5.8

4.3 Program

```
<program>           ::= <program unit list> .

<program unit list> ::= <program unit>
                      | <program unit list> ; <program unit>

<program unit> ::= <main program>
                 | <subpgm>
```

Program units are compiled separately. Subprogram calls are used for communication among program units.

A complete program consisting of a <main program> and all referenced <subpgm>s is called an executable program.

Execution always starts at the first statement of the main program and terminates after the execution of the last statement in the main program.

4.3.1 Main program

```
<main program> ::= MAIN ; <program body> EXIT
```

4.3.2 Subprograms

```
<subpgm>          ::= <fcn>
                    | <subr>

<fcn>             ::= <fcn head> <program body>
                        RETURN ( <returned ϒ expr> )

<subr>            ::= <subr head> <program body> RETURN

<fcn head>        ::= <basic type> FUNCTION <entry list> ;

<subr head>       ::= SUBROUTINE <entry list> ;
                    | SUBROUTINE <entry name> ;

<entry list>      ::= <entry name> ( <param decl list> )

<entry name>      ::= <subpgm identifier>

<returned ϒ expr>     ::= <ϒ expr>
```

A subprogram is defined externally to the program unit that references it. <entry name> is the symbolic name used to reference the subprogram.

A subroutine must be logically and physically terminated by a "RETURN" and that is the only return from the subroutine to the calling program. Similarly a function must be terminated by a "RETURN" followed by the parenthesized expression.

A subprogram may not refer to itself either directly or indirectly through the calling of other subprograms. However, the JOTS compiler doesn't check this. A subroutine can only be referred to in a call statement. A function is referred to by using it's entry name in an expression and following it with the required actual arguments enclosed in parentheses. The type of the returned expression must be assignment compatible with the declared type of the function (see section 4.5.3).

4.3.3 Program body

<program body> ::= <declarations> <stmt list>

4.4 Declarations in a program unit

4.4.1 Declarations

```
<declarations> ::= <decl list> ;
                 | <empty>

<decl list>    ::= <decl list> ; <decl>
                 | <decl>

<decl>         ::= <simple var decl>
                 | <array decl>
                 | <subpgm decl>
                 | <fmt decl>
```

Semantics

Declarations serve to associate identifiers with data types and storage cells.

Every identifier used in a program unit must be defined. An identifier is said to be defined if it satisfies any one of the following:

1) a simple variable - declared in a simple variable declaration
2) an array reference - declared in an array declaration

3) a function - declared in a function declaration

4) a subroutine - declared in a subroutine declaration

5) a format variable - declared in a format declaration

6) a formal parameter - declared in the parameter declaration

7) a label - used as a label

All identifiers, except adjustable dimension variables in a parameter bound list and labels must be defined before they are used. An identifier cannot be defined more than once.

4.4.2 Types

```
<simple type>   ::= <basic type>
                  | STRING ( <string size> )

<basic type>    ::= INTEGER
                  | REAL
                  | LONGREAL
                  | LOGICAL

<string size>   ::= <integer constant>
```

Semantics

String size must be greater than zero.

Integer type

An integer datum may assume a positive, negative, or zero value.

Real and Longreal type

Real and longreal data represent rational numbers which may assume positive, negative, or zero values.

Logical type

A logical datum may assume only the values of true or false.

String type

A string(n) datum is a string of n characters.

4.4.3 Simple variable and array declarations

An array is a named ordered sequence of data. An array element is one member of the sequence of data and is identified by the use of the array name and subscript. The number of dimensions of an array is equal to the number of entries in the dimension list of an array declaration.

```
<simple var decl>   ::= <simple type> <s var list>

<s var list>   ::= <s var>
                 ! <s var list> , <s var>

<s var>        ::= <identifier>
                 ! <identifier> = <init value>

<array decl>   ::= <simple type> ARRAY [ <range list> ]
                   <a var list>

<a var list>   ::= <a var>
                 ! <a var list> , <a var>

<a var>        ::= <identifier>
                 ! <identifier> = <init group>
```

Semantics

Each identifier in the identifier list of a simple variable declaration or an array declaration is associated with a variable or array of type specified by <simple type> and has a scope local to the program unit. The maximum number of dimensions for an integer, real, longreal or logical array is 3 while the maximum for a string array is 2.

Examples

```
INTEGER ARRAY[-2:9] array1, array2
STRING(3) s1, s2 = 'eba'
```

4.4.4 Dimension specification for arrays

```
<range list>   ::= <range>
                 | <range list> , <range>

<range>        ::= <upper range>
                 | <lower range> : <upper range>

<lower range>  ::= <range value>

<upper range>  ::= <range value>

<range value>  ::= <integer constant>
                 | - <integer constant>
```

Semantics

When the lower range is omitted, 1 is assumed. The value of the upper range must be at least that of the lower range.

The size of a dimension is defined as

<upper range> - <lower range> + 1

The size of an array is equal to the product of the sizes of its dimensions.

Examples

```
REAL ARRAY [-20:30, +0] a1,a2
LONGREAL ARRAY [40] x1
```

4.4.5 Initial values

```
<init group>     ::= <init value>
                   ! <repeat factor> ( <init repeat list> )
                   ! ( <init repeat list> )

<init repeat list>  ::= <init group>
                   ! <init repeat list> , <init group>

<init value>     ::= <arith constant>
                   ! - <arith constant>
                   ! <logical constant>
                   ! <string constant>

<repeat factor>      ::= <integer constant>

<arith constant>     ::= <integer constant>
                   ! <real constant>
```

Semantics

Arithmetic constants are promoted to the type of the declared variable, if required. The initial value for a variable must agree in type with the declared variable. Repetition of a sequence of elements may be specified by making the sequence into a list and preceding it by an integer value specifying the number of times the list is to be used.

When initializations for an array are specified, the total number of initial values in the initial term list must not be greater than the size of the array.

Examples

```
    LONGREAL x=3, y = 5E+1, z, w=1.

    LOGICAL ARRAY[0:9, 10] bool = (20(TRUE, FALSE), 25(FALSE),
           5(TRUE, 2(FALSE, 2(TRUE))))

    INTEGER ARRAY[8] count = 8(9)
```

4.4.6 Subprogram declarations

A subprogram declaration declares a variable to be a subprogram name which is referenced within the program unit.

```
<subpgm decl>  ::= <fcn decl>
                 | <subr decl>
```

```
<fcn decl>      ::= EXTERNAL <basic type>
                     FUNCTION <subpgm identifier list>

<subr decl>     ::= EXTERNAL SUBROUTINE
                     <subpgm identifier list>

<subpgm identifier list> ::= <subpgm identifier>
                   | <subpgm identifier list> ,
                     <subpgm identifier>
```

Semantics

Every subprogram is defined externally to the program unit that references it. In a program unit, when a subprogram is to be referenced, the subprogram name must appear in an subprogram declaration.

Each identifier in the identifier list of a function declaration is associated with a function of the type as specified by the <basic type> preceding the reserved word FUNCTION.

Each identifier in the identifier list of a subroutine declaration is associated with a subroutine.

Examples

```
    EXTERNAL INTEGER FUNCTION getnum, random
    EXTERNAL SUBROUTINE decomp, solve, debug
    EXTERNAL REAL FUNCTION f
```

4.4.7 Formal parameter declarations

```
<param decl list>    ::= <param decl>
                       | <param decl list> ; <param decl>

<param decl>     ::= <param var decl>
                   | <param array decl>
                   | <subpgm decl>

<param var decl>     ::= <simple type> <identifier list>

<param array decl>   ::= <simple type> ARRAY [
                         <param bound list> ] <identifier list>
```

Semantics

Each identifier declared in a parameter declaration is called a formal parameter. No initialization is allowed for a formal parameter.

Example

```
SUBROUTINE plot(INTEGER ARRAY[3] a, b; REAL x)

REAL FUNCTION zero(INTEGER a, b, t, eps; EXTERNAL
        REAL FUNCTION f)
```

4.4.8 Dimension specifications for array as a formal parameter

```
<param bound list>  ::= <auto bound list>
                      | <bound list>

<auto bound list>   ::= *
                      | <auto bound list> , *

<bound list>    ::= <bound>
                  | <bound list> , <bound>

<bound>         ::= <range>
                  | <adjustible dimension>

<adjustible dimension>   ::= <identifier>
```

Semantics

The number of asterisks appearing in the auto bound list is the number of dimensions of the array parameter. The use of "*" indicates that the dimension of the array parameter is automatically adjusted according to the dimensions of the actual array argument of the calling program. It is assumed that the corresponding actual argument is an automatic array argument (see section 4.5.4).

The use of range as a bound specification of an array parameter declaration has the same effect and restrictions as decribed in section 4.4.4.

An identifier used as an adjustable dimension must be declared as an integer variable formal parameter. An identifier may appear more than once as an adjustable dimension variable. The calling program passes the specific dimensions of an array to the subprogram via the adjustable dimension formal parameters. Adjustable dimensions can be passed through more than one level of subprogram. The specific dimensions passed to the subprogram as actual arguments should not exceed the true dimensions of the indicated array.

Examples

```
    SUBROUTINE printm(INTEGER m, n; REAL ARRAY[10, 2:10] matrix)
    SUBROUTINE a(REAL ARRAY[n, n] b, c; INTEGER n; REAL x)
    INTEGER FUNCTION search( INTEGER ARRAY[*] number_array;
                        INTEGER m)
```

4.4.9 Format declaration

```
<fmt decl>        ::= FORMAT ( <fmt class> ) <decl fmt list>

<fmt class>       ::= READ | WRITE | PRINT

<decl fmt list>      ::= <decl fmt>
                     | <decl fmt list> , <decl fmt>

<decl fmt>        ::= <identifier> = <fmt group>
```

Semantics

An identifier in a format declaration statement is associated with a variable having a format description specified by <fmt group>. Formats of class READ, WRITE and PRINT can be used only in READ statements, WRITE statements and PRINT statements respectively.

Examples

```
   FORMAT (PRINT) heading = (PAGE, x(20), A(112)),
                  fmt_a = (SKIP(2), F(15, 5, 2));

   FORMAT (READ) input_fmt  = 10(I(2));
```

4.5 <u>STATEMENTS</u>

4.5.1 Statement

```
<stmt list>      ::= <stmt>
                   ! <stmt list> ; <stmt>

<stmt>           ::= <unlabelled stmt>
                   ! <label head> <unlabelled stmt>

<label head>     ::= <identifier> :
                   ! <label head> <identifier> :

<unlabelled stmt>   ::= <compound stmt>
                   ! <assignment stmt>
                   ! <call stmt>
                   ! <goto stmt>
                   ! <if stmt>
                   ! <iterative stmt>
                   ! <read stmt>
                   ! <write stmt>
                   ! <print stmt>
                   ! <rewind stmt>
                   ! <endfile stmt>
                   ! <empty>
```

Semantics

The identifier in the label heading defines the identifier as a label so that it may be referred to in preceding or later GOTO statements. A label identifier has scope local to the program unit. The <empty> in the unlabelled statement denotes a null statement which results in no action during execution.

4.5.2 Compound statement

<compound stmt> ::= BEGIN <stmt list> END

Semantics

A compound statement is a group of statements which can be treated as a single statement. It is especially useful as the object of control statements.

Examples

```
BEGIN
  a := 1;
  b := 2;
  call proc
END
```

4.5.3 Assignment statement

```
<assignment stmt>   ::= <T0 left part> <T1 expr>

<T left part>  ::= <T var> :=
```

Semantics

An assignment statement causes the assignment of the value of the expression to the variable, subjected to the restriction that the type T1 is assignment compatible with the type T0.

A simple type T1 is said to be assignment compatible with a simple type T0 if either

1) the two types are identical
2) T0 and T1 are each either integer, real or longreal
   (this is referred as arithmetic assignment)

Arithmetic assignment

Table 4.1 shows the necessary conversions for the arithmetic assignment statement

<T0 var> := <T1 expr>

T1 \ T0	INTEGER	REAL	LONGREAL
INTEGER	assign	float & assign	float, long & assign
REAL	truncate & assign	assign	long & assign
LONGREAL	short, truncate, & assign	short & assign	assign

Table 4.1

The definitions for truncate, long, short, and float are defined the same as the corresponding built-in functions in section 4.6.6.

String assignment

string1 := string2

The length of string1 must not be less than that of string2. If the length of string1 is greater than the length of string2, the effect is as though string2 were extended to the right with blank characters until it is the same length as string1, and then assigned.

Logical assignment

```
<logical var> := <logical expr>
```

Execution of the logical assignment causes the evaluation of the logical expression and then the resulting value is assigned to the logical variable.

Examples

```
   int_var := 1 + 2.        --- arith assignment
   str := str1              --- string assignment
   bool := true             --- logical assignment
```

4.5.4 Call statement

```
<call stmt>     ::= CALL <subpgm identifier>
                  | CALL <subpgm identifier> ( <arg list> )

<arg list>      ::= <arg>
                  | <arg list> , <arg>

<arg>           ::= <T expr>
                  | <auto array arg>

<auto array arg> ::= <identifier> [ <auto indicator list> ]
```

```
<auto indicator list>    ::= *
                    | <auto indicator list> , *
```

Semantics

The subprogram identifier in the call statement must be declared in a subroutine declaration. The actual arguments in the argument list must agree in order, number, and type with the corresponding formal parameters in the called subroutine. But this is not checked by the compiler. An automatic array argument indicates extra information about the dimensions and lower bounds of the array to be passed automatically to the called subprogram. The corresponding formal parameter should be declared as an automatic adjustable array parameter.

Examples

```
SUBROUTINE proc(INTEGER ARRAY[-1:3] a1, b1; INTEGER c;
    REAL ARRAY[*, *] m; INTEGER d);
 .....
 .....
RETURN;

MAIN;
   INTEGER i;
   EXTERNAL SUBROUTINE proc;
   INTEGER ARRAY[-1:3] a,b;
   INTEGER c,d;
   REAL ARRAY[20, 30] matrix;
   CALL proc(a, b, c, matrix[*, *], d);
    ....
EXIT.
```

The call statement indicates a subroutine call for proc with five arguments. Matrix is an automatic array argument.

4.5.5 Goto statement

```
<goto stmt>    ::= GOTO <label identifier>

<label identifier>  ::= <identifier>
```

Semantics

An identifier is called a label if it appears as a label. A label identifier must be defined as an identifier in the label head of one and only one statement in the same program unit. Execution of a GOTO statement causes the statement identified by the label identifier to be executed next in the execution sequence.

Examples

```
GOTO exit
GOTO loop
```

4.5.6 If statement

```
<if stmt>       ::= <if clause> <stmt>
                  ! <if clause> <stmt> ELSE <stmt>

<if clause>     ::= IF <logical expr> THEN
```

Semantics

The execution of an IF statement causes certain statements to be executed or skipped depending on the value of the <logical expr>.

The form <if clause> <stmt> is executed as follows :

1) the logical expression is evaluated.
2) if the result of the evaluation is true, statement is executed, otherwise no action is taken at all.

The form <if clause> <s1> ELSE <s2> is executed as follows :

1) the logical expression is evaluated.
2) if the result of the evaluation is true s1 is executed and s2 is skipped, otherwise s1 is skipped and s2 is executed.

The two grammar rules for IF statements are clearly ambiguous. The interpretation of these two rules is that each ELSE is associated with the last preceding "un-ELSE'd" IF.

For example

```
   IF c1
   THEN
      IF c2
      THEN s1
      ELSE s2
   ELSE s3
```

"ELSE s3" is associated with "IF c1" and
"ELSE s2" is associated with "IF c2".

Examples

```
   IF a > b
   THEN BEGIN
      IF done
      THEN
         done := FALSE;
      a := b
   END
   ELSE
      b := a;
```

4.5.7 Iterative statement

```
<iterative stmt>     ::= DO <stmt> WHILE <logical expr>
                         <stmt>
```

Semantics

The iterative statement of the form

```
   DO <s1> WHILE c <s2>
```

is exactly equivalent to

```
      BEGIN
   d :  <s1>;
        IF c
```

```
          THEN
              BEGIN <s2> ; GOTO d END
      END
```

Examples

```
    DO WHILE a > b
    BEGIN
        ...
        a := b*a;
        a := a**a
    END
```

works as a while statement.

```
    DO
        BEGIN
          ....
          a := 1
        END
        WHILE NOT DONE ;
```

works as an until statement.

```
    i := 1;
    DO WHILE i <= n
        BEGIN
            a[i] := 0;
            i := i + 1
        END
```

works as a do loop, except for the fact that there may be a 0 trip count.

```
    DO CALL getchar(c) WHILE c ~= 0 CALL putchar(c);
```

4.5.8 Input/output statements

```
<read stmt>      ::= READ ( <input control> ) <io item list>

<input control>      ::= <control head>
                   | <control head> , <input opt list>

<input opt list>     ::= <input opt>
                   | <input opt list> , <input opt>

<input opt>      ::= <eof opt>
                   | <err opt>

<write stmt>     ::= WRITE ( <output control> ) <output data>

<print stmt>     ::= PRINT ( <output control> ) <output data>

<output control>     ::= <control head>
                   | <control head> , <err opt>

<output data>    ::= <io item list>
                   | <empty>

<control head> ::= <file unit> , <fmt specifier>

<file unit>      ::= <identifier>
                   | <integer constant>
```

```
<fmt specifier>      ::= = <fmt group>
                   | <identifier>
                   | *
                   | RECORD

<eof opt>        ::= END = <label identifier>

<err opt>        ::= ERR = <label identifier>

<io item list> ::= <io item>
                   | <io item list> , <io item>

<io item>        ::= <identifier>
                   | <io array>

<io array>       ::= <identifier> [ <range specifier list> ]

<range specifier list>   ::= <range specifier>
                   | <range specifier list> , <range specifier>

<range specifier>    ::= <integer expr>
                   | <integer expr> : <integer expr>
                   | *
```

Semantics

The input/output statements cause a transfer of records between the sequential file identified by <file unit> and internal storage.

A formatted record consists of a sequence of characters that are capable of representation in the processor.

An unformatted record consists of a sequence of values in a processor-dependent form.

FORMAT SPECIFIERS

A RECORD format specifier is used for unformatted records. When a RECORD format specifier is referenced in an I/O statement, the items in the input/output list are transmitted or received directly in the form in which they are stored within the processor. Format specifiers other than RECORD are used for formatted records. An identifier appearing as a format specifier must either be a format variable or an array name. When an array name is referenced in such a manner, the first part of the information contained in the array must constitute a valid Fortran format specification when the I/O statement is executed. An asterisk format specifier indicates a list directed I/O which record control is determined solely by the I/O list.

A READ, WRITE, and PRINT format can only be used in READ, WRITE, and PRINT statements respectively. Definitions and usage of these formats are described in section 4.7.

OPTIONS

An option END = <label identifier> specifies that if the processor encounters an end-of-file condition during the execution of the statement, then the statement identified by the label identifier is to be executed next in the execution sequence.

An option ERR = <label identifier> specifies that if the processor encounters an error condition during the execution of the statement, then the statement identified by the label identifier is to be executed next in the execution sequence.

In an input option list, an <eof opt> or <err opt> can be specified at most once.

ARRAYS IN INPUT/OUTPUT LIST

A range specifier list in an IO array specifies one or more elements of the array to be designated for transmission between core storage and an input or output file. The number of range specifiers in the list must be equal to the number of dimensions of the corresponding array.

The order of transmission of the array elements is determined by the following rules:

1. The n th range specifier in the range specifier list

specifies the range values of the n th subscript of the array variable.

2. The last subscript varies least rapidly, and the first varies most rapidly.
3. The range specifier E1 : E2, where E1 and E2 are integer expressions, specifies the subscript is to be varied from E1 through E2 inclusively.
4. The range specifier E, where E is an integer expression, is the same as E : E.
5. The range specifier * specifies the subscript is to be varied from L through U inclusively, where L and U are the lower bound and upper bound of the subscript of the array.

Examples

```
WRITE(6, RECORD, ERR=write_error) a, b;

READ(card, =20(i(1))) generation[i, 1:size];

If z has been declared as  INTEGER ARRAY[2, -1:2, 2:4] z
then
  PRINT(5, fmt) z[i, 1:2, *];
  implies the transmission of elements of z in the
  following order:
    z[i,1,2], z[i,2,2], z[i,1,3], z[i,2,3],
    z[i,1,4], z[i,2,4]
```

4.5.9 Auxiliary input/output statements

```
<rewind stmt>  ::= REWIND <file unit>

<endfile stmt> ::= ENDFILE <file unit>
```

Semantics

REWIND <file unit>

Execution of this statement causes the unit identified by <file unit> to be positioned at its initial point.

ENDFILE <file unit>

Execution of this statement causes the recording of an endfile record on the unit identified by <file unit>.

Examples

```
REWIND disk_3
ENDFILE tape
```

4.6 Expressions

Expressions are rules which specify how new values are computed from existing ones. The operands are either constants, variables, function calls, or other expressions, enclosed by parentheses if necessary.

In the following sections which describe the syntax of expressions, some of the grammar rules are ambiguous. The ambiguity of these rules is resolved by the specification of binding precedence for operators.

For example

```
a + b*c + d
```

is equilavalent to

```
(a + (b*c)) + d
```

because "+" binds left-right and "*" has a higher precedence than "+" according to table 4.1 in section 4.6.3.

4.6.1 Type promotion

The precedence of types in increasing order is INTEGER, REAL AND LONGREAL.

<u>ARITHMETIC</u> <u>type</u> <u>promotion</u> for two or more arithmetic operands

If T is the type with highest precedence for all operands, the operands with lower precedence type are converted into type T.

4.6.2 Variables

```
<T var>              ::= <identifier>
                        | <subpgm identifier> ( <arg list> )
                        | <T array designator>

<T array designator> ::= <identifier> [ <subscript list> ]

<subscript list>     ::= <subscript>
                        | <subscript list> , <subscript>

<subscript>          ::= <integer expr>
```

Semantics

Argument list has been defined in section 4.5.4.

An array designator denotes the variable whose indices are the values of the integer expressions in the subscript list. Each subscript must be an integer expression whose value lies within the declared bounds for that subscript position. However, no code is generated to detect violations of array bounds.

4.6.3 Integer, Real and Longreal expressions

```
<T0 expr>      ::= <T0 var>
                 | <built-in T0 fcn>
                 | <T0 constant>
                 | - <T0 expr>
                 | + <T0 expr>
                 | ( <T0 expr> )
                 | <T1 expr> <arith op> <T2 expr>

<arith op>     ::= **  |  %  |  *  |  /  |  +  |  -
```

Semantics

When a syntactic rule has the symbol T0 on both sides, T0 has to be replaced by the word integer, real or longreal. When the symbols T0, T1 and T2 appear in a syntactic rule, they have to be replaced by any combination of words according to table 4.2 which indicates T0 for any combination of T1 and T2.

T1 \ T2	INTEGER	REAL	LONGREAL
INTEGER	INTEGER	REAL	LONGREAL
REAL	REAL	REAL	LONGREAL
LONGREAL	LONGREAL	LONGREAL	LONGREAL

Table 4.2

The order in which the indicated operations are performed depends on the precedence of the operators appearing in the arithmetic expression, unless the order is changed by the use of parentheses.

Table 4.3 shows the precedence and binding of the arithmetic operators.

OPERATOR	MEANING	BINDING
-	unary minus	
+	unary plus	
**	exponentiation	right-left
%	modulus	left-right
*	multiple	left-right
/	divide	
+	binary add	left-right
-	binary subtract	

Table 4.3 Precedence and binding of arithmetic operators.

The precedence of the operators in table 4.3 decreases from top to bottom with operators in the same row having the same precedence.

Examples

```
- a

-a + b * c + d

  is equivalent to ((-a) + (b*c)) + d

d % a ** b ** (c + 1)
```

is equivalent to d%(a**(b**(c + 1)))

4.6.4 String expressions

```
<string expr>  ::= <string var>
                 | <string constant>
```

4.6.5 Logical expressions

```
<logical expr> ::= <logical element>
                 | <T1 expr> <relational op> <T2 expr>
                 | NOT <logical expr>
                 | <logical expr> AND <logical expr>
                 | <logical expr> OR <logical expr>
                 | ( <logical expr> )

<logical element>   ::= <logical var>
                 | <logical constant>

<logical constant>  ::= TRUE
                 | FALSE

<relational op>     ::= > | < | >= | <=  | = | ~=
```

Semantics

Table 4.4 shows the precedence and binding of the logical operators.

OPERATORS	BINDING
NOT	
relational operators	
AND	left-right
OR	left-right

Table 4.4 Precedence and binding of logical operators.

The precedence of the logical operators in table 4.4 decreases from top to bottom.

The relational operation

<T1 expr> <relational op> <T2 expr>

is compatible if either

1) both T1 and T2 are of string type.

2) both T1 and T2 are either integer, real or longreal, but not necessarily the same.

In case 1 of the above, if the operands are of unequal length, the shorter operand is considered as if it were extended on the right with blanks to length of the longer operand.

In case 2 of the above arithmetic type promotion (section 4.6.1) is performed.

The resulting type of a valid logical expression is always logical.

Examples

```
'ab' > string1

a + b*c > 1.E0 OR b >c AND NOT bool

   is equivalent to

    ((a + (b*c)) > 1.E0)  OR  ((b > c) AND (NOT bool))
```

4.6.6 Built-in functions

```
<built-in τ fcn>     ::= <one-arg τ fcn>
                       | <many-arg τ fcn>
```

4.6.6.1 Built-in functions with one argument

```
<one-arg τ0 fcn>     ::= <one-arg fcn name> ( <τ1 expr> )

<one-arg fcn name>   ::= SHORT | FLOAT | LONG
                       | TRUNCATE | ROUND | FLOOR | CEILING
                       | EXP | LOG | LOG10 | SIN | COS | ATAN
                       | SIGN | SQRT
```

Semantics

The symbol τ0 denotes the resulting type and τ1 the argument type of a built-in function. The replacement words for these symbols and the definitions of the built-in functions are specified individually in each of the following sections.

SHORT

Argument type : longreal

Result type : real

Definition :

obtain most significant part of the longreal argument.

LONG

Argument type : real

Result type : longreal

Definition :

express real argument in longreal form.

FLOAT

Argument type : integer

Result type : real

conversion from integer to real

ABS

Argument type : integer, real or longreal

Result type : same as the type of argument

Definition :

ABS(x) is |x|

SIGN

Argument type : integer, real or longreal

Result type : integer

Definition :

```
   SIGN(x) is -1 if x < 0
               0 if x = 0
               1 if x > 0
```

TRUNCATE, ROUND, FLOOR, and CEILING

TRUNCATE, ROUND, FLOOR and CEILING accept an argument of type real or longreal. The resulting type is always integer. The definitions of these functions are as follow :

TRUNCATE(x)	sign of x times largest integer <= ABS(x)
ROUND(x)	TRUNCATE(x + SIGN(x)*0.5)
FLOOR(x)	largest integer <= x
CEILING(x)	smallest integer >= x

Other functions

EXP, LOG, LOG10, COS, SIN, ATAN, and SQRT accept an argument of type integer, real or longreal. The resulting type is the same as that of the argument. The definitions of these functions are as follow:

EXP(x)	e**x
LOG(x)	natural logarithm (base e) of x
LOG10(x)	common logarithm (base 10) of x
SIN(x)	trigonometric sine of x (x in radians)
COS(x)	trigonometric cosine of x (x in radians)
ATAN(x)	arctan of x (result in radian)
SQRT(x)	square root of x

4.6.6.2 Built-in functions with two or more arguments

```
<many-arg τ fcn>      ::= MAX ( <many-arg list> )
                       | MIN ( <many-arg list> )

<many-arg list>       ::= <τ1 expr> , <τ2 expr>
                       | <many-arg list> , <τ3 expr>
```

Semantics

The symbols $\tau 1$, $\tau 2$, and $\tau 3$ have to be replaced by one of the words "integer", "real" or "longreal".

These functions accept at least two arguments and all arguments must be integer, real or longreal type. Arithmetic type promotion (section 4.6.1) is performed for the arguments. The resulting type τ is the same as the arguments having type of highest precedence.

MAX returns the value of the argument with largest value.

MIN returns the value of the argument with smallest value.

4.7 Input/output formats

```
<fmt group>        ::= <edit desc>
                     ! SKIP ( <integer constant> )
                     ! SKIP
                     ! X ( <integer constant> )
                     ! PAGE
                     ! <string constant>
                     ! <integer constant> ( <fmt repeat list> )
                     ! ( <fmt repeat list> )

<fmt repeat list>    ::= <fmt group>
                     ! <fmt repeat list> , <fmt group>

<edit desc>       ::= <fractional field> ( <fractional desc> )
                     ! I ( <field width> )
                     ! A ( <field width> )
                     ! L ( <field width> )

<fractional field>  ::= D ! E ! F ! G

<fractional desc> ::= <field width> , <fractional digit> ,
                      <scale factor>
                     ! <field width> , <fractional digit>

<field width>  ::= <integer constant>

<fractional digit>  ::= <integer constant>
```

```
<scale factor> ::= <integer constant>
                 | - <integer constant>
```

Semantics

Repetition of a sequence of format items may be specified by making the sequence into a list enclosed in parentheses and preceding it by an integer value specifying the number of times the list is to be used.

Formats in READ statements or READ format declarations cannot have PAGE and STRING format items. Formats in WRITE statements or WRITE format declarations cannot have PAGE format items. When a format is used in a PRINT statement or a PRINT format declaration, the record transfer through the conversion of the format is assumed for printing purpose. The first character of records thus produced is for vertical spacing during printing.

SKIP descriptor

SKIP(n) indicates n records are to be skipped for the input or output. "SKIP" is equivalent to "SKIP(1)". The appearance of SKIP(0) in a format for READ and WRITE statements has no effect, and its appearance in a format for PRINT statements causes no advance in the vertical spacing before printing.

X descriptor

X(n) indicates the transmission of the next character to or from a record is to occur at the position n characters from the current position.

PAGE descriptor

PAGE causes an advance to first line of next page before printing of the next character to or from a record.

STRING constant

A string constant in a format causes the characters of the string constant to be written out.

A descriptor

The A(n) descriptor causes n characters to be read into, or written from, the specified list element. The corresponding element should be a string(n) variable or array element.

L descriptor

The L(w) descriptor indicates that the external field occupies w position as a string of infomation for a logical datum. The external input field must consist of optional blanks followed by a T or F followed by optional characters, for true or false, respectively. The external output field consists of w - 1 blanks followed by a T or F as the value of the internal datum is true or false, respectively.

Numeric editing

To each numeric descriptor in a format specification, there corresponds one element specified by the input/output list.

When the scale factor is omitted in a fractional descriptor, zero is assumed. The scale factor n affects the appropriate conversions in the following manner:

1) For F, E, G, and D input conversions (provided no exponent exists in the external field) and F output conversions, the scale factor effect is as follows:

   externally represented number equals internally represented number times the quantity ten raised to the n th power.

2) For F, E, G, and D input, the scale factor has no effect if there is an exponent in the external field.

3) For E and D output, the basic real constant part of the output quantity is multiplied by 10**n and the exponent is reduced by n.

4) For G output, the effect of the scale factor is suspended unless the magnitude of the datum to be converted is outside the range that permits the effective use of F conversion.

The I descriptor is used to specify input/output of integer data. The numeric field descriptors F, E, G, and D are used to specify input/output of real, and longreal data.

1) With all the numeric input conversions, leading blanks are not significant and other blanks are zero. Plus signs may be omitted. A field of all blanks is considered to be zero.
2) With the F, E, G, and D input conversions, the format of an input field must have the same form as a <real constant> (section 4.2.2). A decimal point appearing in the input field overrides the decimal point specification supplied by the field descriptor.
3) With all output conversions, the output field is right justified. If the number of characters produced by the conversion is smaller than the field width, leading blanks will be inserted in the output field.
4) With all output conversion, the external representation of a negative value must be signed; a positive value may be signed.
5) The number of characters produced by an output conversion must not exceed the field width.

Examples

```
(A(3), skip(2), E(15, 3, 2), X(2))
2(I(4))
```

4.8 Manifests, comments and compiler options

4.8.1 Manifests

```
<manifest decl>      ::= # <identifier> <manifest defn>
                               <manifest delim>

<manifest delim>     ::= <newline> | !
```

where <manifest defn> is a string of characters containing no <newline> or "!".

Semantics

A manifest declaration is restricted to be outside all program units. The "#" in a manifest declaration must be the first character of the source line in which the manifest is declared.

Certain manifests have been pre-defined by the translator and they are called pre-defined manifests. Appendix C gives a listing of the pre-defined manifests.

A manifest identifier is said to be defined if it is declared in a manifest declaration or it is a pre-defined manifest. Only pre-defined manifests can be redefined. A manifest has a scope global to the program. Every manifest must be defined before used.

The appearance of a manifest in a program unit has the effect of a textual expansion of the manifest definition in the place where the manifest name appears. The expansion of a manifest results in expansions of other manifests if their names appear in the definition. Recursive manifest definitions are not permitted.

Examples

```
# do_forever  do while true
```

declares "do_forever" as manifest with "do while true" as its definition.

```
# list write(printer, *)
#printer 8 ! unit 8 for printer
```

declares "printer" as a manifest with a definition of "8 ", and "list" as a manifest with definition "write(printer, *)". When "list" is referred in the program unit, the final expansion is 'write(8 , *)".

4.8.2 Comments

Comments are removed from the token stream in the input phase and transliterated as comments in the object Fortran program.

```
<comment>      ::= ! <string body> <newline>
```

where <newline> is the ASCII new line character.

Programmers can have some control over the format of the transcribed comments. In the source, a comment starting at the first column of a line is transcribed as is with characters beyond column 72 truncated. Comments starting beyond the first column are transcribed with indentation and continued on the next line if required.

4.8.3 Compiler Options

The compiler has various options available for the user; for example, source listing and symbol table dump. There is a set of default options, but each option may be set by using a control toggle. An option x can be turned on or turned off by having "$x" or "$~x" respectively immediate after the comment character "!".

Table 4.5 is a listing of the options.

OPTION	USE	DEFAULT
C	emit comments	on
D	symbol table dump	off
L	source listing	on
T	scanner output	off

Table 4.5 Compiler options

4.9 Sample programs

The Fortran outputs of the examples given in this section have passed the PFORT verifer and have been run the the Honeywell 6060.

4.9.1 The function ZEROIN

A JOTS version of the function ZEROIN given by Brent (1973, p.189) is shown here as an example. The Algol version of ZEROIN and the Fortran translation of it as given in Brent (1973) can be found in appendix B.

Source listing from translator:

```
*** JOTS COMPILER Version 1 Level 0 (MAY 76)***
                        TODAY IS 06/22/76     TIME 23:13:22

 line        -------- input --------

       |!
       |!$d
       |real function zeroin(real a, b, t;
       |                          external real function f);
       |   real c, d, e, fa, fb, fc, tolerance, m, p, q, r, s;
       |   fa := f(a);  fb := f(b);
       |
       |   int :
       |   c := a; fc := fa; d := b - a;  e := d;
       |
       |   ext :
       |   if abs(fc) < abs(fb)
       |   then begin
       |      a := b; b:= c; c := a;
       |      fa := fb; fb := fc; fc := fa
       |   end;
       |   tolerance := 2*machine_eps*abs(b) + t;
       |   m := 0.5*(c - b);
       |   if abs(m) > tolerance  and fb ~= 0
       |   then begin  ! see if bisection is forced
       |      if abs(e) < tolerance  or  abs(fa) <= abs(fb)
       |      then begin
       |         d := m;
       |         e := m
       |      end
       |      else begin
       |         s := fb/fa;
       |         if a = c
       |         then begin ! Linear interpolation
|              p := 2*m*s;
|              q := 1 - s
|           end
|           else begin   ! Inverse quadratic interpolation
|              q := fa/fc;
|              r := fb/fc;
|              p := s*(2*m*q*(q-r) - (b - a)*(r-1));
|              q := (q - 1)*(r - 1)*(s - 1)
|           end;
|           if p > 0
|           then
|              q := -q
|           else
|              p := -p;
|           s := e; e := d;
|           if 2*p < 3*m*q - abs(tolerance*q)
|              and p < abs(0.5*s*q)
|           then
|              d := p/q
|           else begin
|              d := m;
|              e := m
|           end
|        end;
|        a := b; fa := fb;
|        if abs(b) > tolerance
|        then
|           b := b + d
|        else
|           if m > 0
|           then
|              b := b + tolerance
|           else
|              b := b - tolerance;
|        fb := f(b);
|        if fb > 0 and fc > 0  or  fb <= 0 and fc <= 0
|        then
|           goto int;
|        goto ext;
|     end;
|return(b).
```

```
*** SYMBOL TABLE DUMP ***

        name              fortran name          type      class

   ZEROIN                ZEROIN                real      entry
   A                     A                     real
   B                     B                     real
   T                     T                     real
   F                     F                     real      subprogram
   C                     C                     real
   D                     D                     real
   E                     E                     real
   FA                    FA                    real
   FB                    FB                    real
   FC                    FC                    real
   TOLERANCE             TOLAE                 real
   M                     M                     real
   P                     P                     real
   Q                     Q                     real
   R                     R                     real
   S                     S                     real
   INT                   101                   label
   EXT                   102                   label
```

```
C
C$d
C
      REAL   FUNCTION ZEROIN(A, B, T, F)
        LOGICAL STREQ, STRNE, STRGT, STRGE, STRLT, STRLE
        REAL    A, B, T
        REAL    F
        EXTERNAL F
        REAL    C, D, E, FA, FB, FC, TOLAE, M, P, Q, R, S
C
        FA = F(A)
        FB = F(B)
C
  101   CONTINUE
        C = A
        FC = FA
        D = B - A
        E = D
C
  102   CONTINUE
        IF                                         ( .NOT.
          (ABS(FC) .LT. ABS(FB))                    ) GO TO 5001
C        ..THEN
            A = B
```

```
               B = C
               C = A
               FA = FB
               FB = FC
               FC = FA
 5001      CONTINUE
C
           TOLAE = ((2.*0.74506E-8)*ABS(B)) + T
           M = 0.5*(C - B)
           IF                                             ( .NOT.
     .       ((ABS(M) .GT. TOLAE) .AND. (FB .NE. 0.))        ) GO TO 5002
C          ..THEN
C              ... see if bisection is forced
               IF                                         ( .NOT.
     .           ((ABS(E) .LT. TOLAE) .OR. (ABS(FA) .LE. ABS(FB)))
     .                                                   ) GO TO 5003
C              ..THEN
                   D = M
                   E = M
                                                            GO TO 5004
 5003          CONTINUE
C              ..ELSE
                   S = FB/FA
                   IF                                     ( .NOT.
     .               (A .EQ. C)                           ) GO TO 5005
C                  ..THEN
C                      ... Linear interpolation
                       P = (2.*M)*S
                       Q = 1. - S
                                                            GO TO 5006
 5005              CONTINUE
C                  ..ELSE
C                      ... Inverse quadratic interpolation
                       Q = FA/FC
                       R = FB/FC
                       P = S*((((2.*M)*Q)*(Q - R)) - ((B - A)*(R - 1.)
     .                   ))
                       Q = ((Q - 1.)*(R - 1.))*(S - 1.)
 5006              CONTINUE
C
                   IF                                     ( .NOT.
     .               (P .GT. 0.)                         ) GO TO 5007
C                  ..THEN
                       Q = -Q
                                                            GO TO 5008
 5007              CONTINUE
C                  ..ELSE
                       P = -P
 5008              CONTINUE
C
                   S = E
                   E = D
```

```
                 IF                                           ( .NOT.
     .             (((2.*P) .LT. (((3.*M)*Q) - ABS(TOLAE*Q))) .AND.
     .             (P .LT. ABS((0.5*S)*Q)))               ) GO TO 5009
C                ..THEN
                     D = P/Q
                                                                GO TO 5010
 5009            CONTINUE
C                ..ELSE
                     D = M
                     E = M
 5010            CONTINUE
C
 5004        CONTINUE
C
             A = B
             FA = FB
             IF                                               ( .NOT.
     .         (ABS(B) .GT. TOLAE)                             ) GO TO 5011
C            ..THEN
                 B = B + D
                                                                GO TO 5012
 5011        CONTINUE
C            ..ELSE
                 IF                                           ( .NOT.
     .             (M .GT. 0.)                                ) GO TO 5013
C                ..THEN
                     B = B + TOLAE
                                                                GO TO 5014
 5013            CONTINUE
C                ..ELSE
                     B = B - TOLAE
 5014            CONTINUE
C
 5012        CONTINUE
C
             FB = F(B)
             IF                                               ( .NOT.
     .         (((FB .GT. 0.) .AND. (FC .GT. 0.)) .OR. ((FB .LE. 0.)
     .          .AND. (FC .LE. 0.)))                          ) GO TO 5015
C            ..THEN
                 GO TO 101
 5015        CONTINUE
C
             GO TO 102
 5002    CONTINUE
C
         ZEROIN = B
         RETURN
       END
```

4.9.2 Game of life

Source listing from translator:

```
*** JOTS COMPILER Version 1 Level 0 (MAY 76)***
                          TODAY IS 05/27/76     TIME  0:29:26

         -------- input --------

|!$DUMP
|#card 5
|#death 0
|#alive 1
|!  game of life
|
|main;
|   integer array[0:11, 0:11] generation, next_generation;
|   integer i, j, m, number_of_generation, number_of_nbr;
|   integer board_size;
|   integer ip1, im1, jp1, jm1;
|   external subroutine input, output, clear, copy;
|
|   read(card, =2(i(2))) board_size, number_of_generation;
|   call clear(board_size, generation[*,*]);
|   call clear(board_size, next_generation[*,*]);
|   call input(board_size, generation[*,*]);
|   print(printer, =('original pattern:', skip(2)));
|   call output(board_size, generation[*,*]);
|!
|   m := 1;
|   do while m <= number_of_generation
|   begin
|      i := 1;
|      do while i <= board_size
|      begin
|         ip1 := i + 1;
|         im1 := i - 1;
|         j := 1;
|         do while j <= board_size
|         begin
|            ! find neighbours of cell(i,j)
|            jp1 := j + 1;
|            jm1 := j - 1;
|            number_of_nbr :=
|               generation[im1,jm1] + generation[im1,j] +
|               generation[im1,jp1] + generation[i,jm1] +
|               generation[i,jp1] + generation[ip1,jm1] +
|               generation[ip1,j] + generation[ip1,jp1];
|            !assume death for next generation
|            next_generation[i,j] := death;
|              if (generation[i,j] = death
|                 and number_of_nbr = 3)
|              then
|                 next_generation[i,j] := alive
|              else
|                 if generation[i,j] = alive and
|                    (number_of_nbr = 2 or number_of_nbr = 3)
|                 then
|                    next_generation[i, j] := alive;
|              j := j + 1;
|           end;
|           i := i + 1;
|        end;
|        print(printer, =(skip(2), 'generation', i(3), ':')) m;
|        call output(board_size, next_generation[*, *]);
|        call copy(board_size, generation[*,*],
|           next_generation[*,*]);
|        m := m + 1;
|     end;
|exit.
```

```
*** SYMBOL TABLE DUMP ***

           name                        fortran name   type         class
--------------------------------------------------------------------------


GENERATION                             GENTN          integer      array
NEXT_GENERATION                        NEXGN          integer      array
I                                      I              integer
J                                      J              integer
M                                      M              integer
NUMBER_OF_GENERATION                   NUMGN          integer
NUMBER_OF_NBR                          NUMNR          integer
BOARD_SIZE                             BOASE          integer
IP1                                    IP1            integer
IM1                                    IM1            integer
JP1                                    JP1            integer
JM1                                    JM1            integer
INPUT                                  INPUT          subroutine   subprogram
OUTPUT                                 OUTPUT         subroutine   subprogram
CLEAR                                  CLEAR          subroutine   subprogram
COPY                                   COPY           subroutine   subprogram
```

Object Fortran code from JOTS compiler:

```
C
C$dump
C  game of life
C
C       MAIN...
         LOGICAL STREQ, STRNE, STRGT, STRGE, STRLT, STRLE
         INTEGER GENTN(12, 12), NEXGN(12, 12)
         INTEGER I, J, M, NUMGN, NUMNR
         INTEGER BOASE
         INTEGER IP1, IM1, JP1, JM1
         EXTERNAL INPUT, OUTPUT, CLEAR, COPY
C
         READ(5, 9001) BOASE, NUMGN
 9001    FORMAT(2(I2))
         CALL CLEAR(BOASE, GENTN,  - 1, 12,  - 1, 12)
         CALL CLEAR(BOASE, NEXGN,  - 1, 12,  - 1, 12)
         CALL INPUT(BOASE, GENTN,  - 1, 12,  - 1, 12)
         WRITE(6, 9002)
 9002    FORMAT(1H ,  17Horiginal pattern:/1H0)
         CALL OUTPUT(BOASE, GENTN,  - 1, 12,  - 1, 12)
C
         M = 1
C
 5001    CONTINUE
C         DO..
C         ...WHILE
                                                      IF (.NOT.
     .           (M .LE. NUMGN)                         ) GO TO 5002
            I = 1
C
 5003       CONTINUE
C            DO..
C            ...WHILE
                                                      IF (.NOT.
     .              (I .LE. BOASE)                     ) GO TO 5004
               IP1 = I + 1
               IM1 = I - 1
               J = 1
C
 5005          CONTINUE
C               DO..
C               ...WHILE
                                                      IF (.NOT.
     .                 (J .LE. BOASE)                  ) GO TO 5006
C                 ... find neighbours of cell(i,j)
                  JP1 = J + 1
                  JM1 = J - 1
                  NUMNR = ((((((GENTN(IM1 + 1, JM1 + 1) + GENTN(
     .               IM1 + 1, J + 1)) + GENTN(IM1 + 1, JP1 + 1))
     .                + GENTN(I + 1, JM1 + 1)) + GENTN(I + 1, JP1
     .                    + 1)) + GENTN(IP1 + 1, JM1 + 1)) + GENTN(
     .                   IP1 + 1, J + 1)) + GENTN(IP1 + 1, JP1 + 1)
C                       ... assume death for next generation
                        NEXGN(I + 1, J + 1) = 0
                        IF                            ( .NOT.
     .                    ((GENTN(I + 1, J + 1) .EQ. 0) .AND. (NUMNR
     .                     .EQ. 3))                   ) GO TO 5007
C                       ..THEN
                           NEXGN(I + 1, J + 1) = 1
                                                        GO TO 5008
 5007                   CONTINUE
C                       ..ELSE
                           IF                         ( .NOT.
     .                       ((GENTN(I + 1, J + 1) .EQ. 1) .AND. ((
     .                       NUMNR .EQ. 2) .OR. (NUMNR .EQ. 3)))
     .                                                ) GO TO 5009
C                          ..THEN
                              NEXGN(I + 1, J + 1) = 1
 5009                      CONTINUE
C
 5008                   CONTINUE
C
                        J = J + 1
                     GO TO 5005
 5006                CONTINUE
C
                     I = I + 1
                  GO TO 5003
 5004             CONTINUE
C
                  WRITE(6, 9003) M
 9003             FORMAT(1H /1H0, 10Hgeneration, I3,  1H:)
                  CALL OUTPUT(BOASE, NEXGN, - 1, 12, - 1, 12)
                  CALL COPY(BOASE, GENTN, - 1, 12, - 1, 12, NEXGN, - 1,
     .              12, - 1, 12)
                  M = M + 1
               GO TO 5001
 5002          CONTINUE
C
               STOP
            END
```

4.9.3 Other examples

```
real function cmod( real x, y );  ! modulus of z = x + iy
   real u, v, modulus_of_z;
   u := max( x, y );
   v := min( x, y );
   if v=0 then  modulus_of_z := u
   else  modulus_of_z := 1.5*u*sqrt( 1/1.5**2 +
                              (v/(1.5*u))**2 );
   return( modulus_of_z )
.
```

```
C
C       ... modulus of z = x + iy
C
      REAL     FUNCTION CMOD(X, Y)
        LOGICAL STREQ, STRNE, STRGT, STRGE, STRLT, STRLE
        REAL     X, Y
        REAL     U, V, MODSZ
C
        U = AMAX1(X, Y)
        V = AMIN1(X, Y)
        IF                                           (  .NOT.
          (V .EQ. 0.)                                ) GO TO 5001
C        ..THEN
             MODSZ = U
                                                        GO TO 5002
 5001   CONTINUE
C        ..ELSE
             MODSZ = (1.5*U)*SQRT((1./(1.5**2.)) + ((V/(1.5*U))**2.))
 5002   CONTINUE
C
        CMOD = MODSZ
        RETURN
      END
 INTEGER FUNCTION month(string(10) month_str);
    string(10) ARRAY[12] month_name = ('JANUARY', 'FEBRUARY',
        'MARCH', 'APRIL', 'MAY', 'JUNE', 'JULY', 'AUGUST',
        'SEPTEMBER', 'OCTOBER', 'NOVEMBER', 'DECEMBER');
    integer i;
    i := 12;
    DO WHILE (month_str ~= month_name[i])
    BEGIN
        i := i - 1;
        if i = 0
        then
            goto out;
    END;
    out:
 RETURN(i);
 main;
    string(10) m;
    integer i;
    external integer function month;
    do
        read(5, =a(10)) m
        while m ~= 'END'
        begin
            i := month(m);
            print(printer, =(i(2))) i;
        end;
 exit.
```

```
C
C
      INTEGER FUNCTION MONTH(MONSR)
        LOGICAL STREQ, STRNE, STRGT, STRGE, STRLT, STRLE
        INTEGER MONSR(3)
        INTEGER MONNE(3, 12)
        INTEGER I
        DATA MONNE(1,1),MONNE(2,1),MONNE(3,1),MONNE(1,2),MONNE(2,2),
     .     MONNE(3,2),MONNE(1,3),MONNE(2,3),MONNE(3,3),MONNE(1,4),
     .     MONNE(2,4),MONNE(3,4),MONNE(1,5),MONNE(2,5),MONNE(3,5),
     .     MONNE(1,6),MONNE(2,6),MONNE(3,6),MONNE(1,7),MONNE(2,7),
     .     MONNE(3,7),MONNE(1,8),MONNE(2,8),MONNE(3,8),MONNE(1,9),
     .     MONNE(2,9),MONNE(3,9),MONNE(1,10),MONNE(2,10),MONNE(3,10),
     .     MONNE(1,11),MONNE(2,11),MONNE(3,11),MONNE(1,12),MONNE(2,12)
     .     ,MONNE(3,12)/4HJANU,4HARY ,4H    ,4HFEBR,4HUARY,4H    ,4
     .     HMARC,4HH   ,4H    ,4HAPRI,4HL   ,4H    ,4HMAY ,4H    ,4
     .     H    ,4HJUNE,4H    ,4H    ,4HJULY,4H    ,4H    ,4HAUGU,4
     .     HST ,4H    ,4HSEPT,4HEMBE,4HR   ,4HOCTO,4HBER ,4H    ,4
     .     HNOVE,4HMBER,4H    ,4HDECE,4HMBER,4H    /
C
```

```
       I = 12
C
 5001  CONTINUE
C       DO..
C       ...WHILE
                                                IF (.NOT.
     .      (STRNE(MONSR, 3, MONNE(1, I), 3))       ) GO TO 5002
           I = I - 1
           IF                                   (  .NOT.
     .      (I .EQ. 0)                         ) GO TO 5003
C          ..THEN
              GO TO 101
 5003      CONTINUE
C
       GO TO 5001
 5002  CONTINUE
C
C
  101  CONTINUE
       MONTH = I
       RETURN
      END
C
C
C      MAIN...
       LOGICAL STREQ, STRNE, STRGT, STRGE, STRLT, STRLE
       INTEGER M(3)
       INTEGER I
       INTEGER MONTH
       EXTERNAL MONTH
C
C
 5001  CONTINUE
C       DO..
           READ(5, 9001) (M(KTEMP), KTEMP = 1, 3)
 9001      FORMAT(2(A4), A2)
C       ...WHILE
                                                IF (.NOT.
     .      (STRNE(M, 3,   3HEND, 1))             ) GO TO 5002
           I = MONTH(M)
           WRITE(6, 9002) I
 9002      FORMAT(1H , I2)
       GO TO 5001
 5002  CONTINUE
C
       STOP
      END
```

5. Discussion of problems

The design objectives for JOTS and its compiler gave rise to many implementation problems. The Fortran code generated from the preprocessor conforms closely to the ANSI Fortran standard. It turns out that it is more difficult to emit Fortran code than to emit machine code. This chapter attempts to discuss the problems involved in the implementation. Solutions to these problems are outlined in Chapter 6.

5.1 Name mapping

Identifiers in the source program must be mapped into Fortran names that are unique within the program unit; the mapping should preserve "readability" of the identifiers. Uniqueness of the name mapping can be easily achieved if the requirement for mapping into "readable" names is ignored. For example, the set {A1, A2, ..., A9, A10, ..., A100, A101, ..., A99999} could be used as the range of the mapping; an algorithm for such a mapping is trivial but the resulting object Fortran code is unreadable.

Identifiers in the source program can have lengths up to the length of a source line while those in Fortran programs are restricted to 6 characters or less. Also, identifiers in the source program may use the character "_". The name mapping must remove these characters since "_" is not allowed in a Fortran identifier.

The algorithm discussed in section 6.3.2 generates legal, unique Fortran identifiers which resemble the source identifiers closely enough to remain "readable".

5.2 Generation of temporary integer variables

Two kinds of integer variables have to be generated:

1) for the "non-standard" subscript expression
2) for the dimensions and offsets of automatic array formal parameters

The generated names should be "readable" and unique within a program unit. The number of temporary variables required may be large. Generating temporary variables for non-standard subscript expression is difficult since a variable has to be generated and an assignment statement has to be emitted to assign the expression to the variable. The assignment statement has to be emitted first and there may be any number of such assignment statements.

The solutions to these problems are discussed in section 6.3.2.

5.3 Declarations

Fortran program units are compiled separately. Subprogram names are stored as symbolic linkage information which is used when the compiled Fortran program units are bound together. For this reason, an external subprogram name in JOTS has the same restriction as a Fortran identifier and the same name is used as the subprogram name in the object Fortran code.

A problem arises when an identifier is mapped into a name which is the same as an external subprogram name in a later declaration statement of the same program unit.

An error message can simply be printed when this kind of situation happens. This is not quite acceptable as long as other ways can be found to solve the problem without much effort.

To impose restrictions on the order of the declarations does not help either. The formal parameters and possibly the temporary variables for automatic array parameters come before all declarations.

A built-in function in the source language may accept different argument types. This means that a built-in function with a certain argument type has to be mapped into a Fortran function name which is either an intrinsic function or a function defined by the translator.

Section 6.3.3 gives an algorithm for solving this problem.

5.4 Arrays

Array offset

In JOTS, an array dimension may have a lower bound other than one. A formal array parameter can be declared as automatic for which temporary integer variables are generated to receive information about the offsets and dimensions of the actual array argument.

Unfortunately, arrays in Fortran can only have an implicit lower bound of one. This means that when an array element is referenced in the source program, an offset may have to be emitted. The offset may be a positive integer constant, a negative integer constant or a name.

Section 6.3.1 describes the implementation and data structure of an array descriptor to allow offsets to be emitted efficiently.

Non-standard subscript expressions

According to the ANSI Fortran standard, a subscript expression can only be written as one of the following constructs:

```
c*v + k
c*v - k
c*v
v + k
v - k
v
k
```

where c and k are integer constants and v is an integer variable reference.

Imposing the same restriction for subscript expressions in JOTS would make the syntax of the language ugly and hard to remember. In addition, offsets may have to be emitted for the subscript which may make the resulting subscript expression non-standard.

The recursive definition of a subscript expression in JOTS makes checking for the non-standard subscripts complicated. It is difficult for this to be checked in the syntactic level.

Section 6.3.6 gives a solution to the above problem.

5.5 Type promotions and type checking

ANSI Fortran only allows mixed mode operations between a real and a double precision operands in an expression. The use of an intrinsic function requires the programmer to choose a different function name for a different argument type.

PROBLEMS

Type promotion (section 4.6.1) is one of the most useful features in JOTS. Arithmetic type promotion promotes operands to the highest precedence type of all operands. Appropriate code is generated for built-in functions accepting different argument types.

The type promotion requirement makes it impossible to emit code for an expression as it is parsed. The Fortran code for an expression must be emitted after the entire source expression has been parsed.

Section 6.3.6 describes the technique used to handle type promotions and type checking.

5.6 Comments

In a Fortran program, the letter "C" in column 1 of a line designates that line is a comment line. A comment line must be immediately followed by an initial line, another comment line, or an end line.

The transliteration of comments from a JOTS program to object Fortran code is an important factor to make the latter readable.

In a JOTS program, the appearance of the special comment character in a line signifies the rest of the line to be treated as a comment. Thus a comment may appear at the middle of a statement.

PROBLEMS

The restrictions on the comment lines in Fortran makes it impossible to transcribe a comment to the object code as it is scanned. One way of doing this is to queue comment lines and dump the queue before every initial line of a Fortran statement is emitted. This is inefficient when there are a lot of comments in the source program.

Section 6.3.8 describes how comments are handled to allow efficient transliteration.

5.7 Strings

In Standard Fortran, the string handling facility is very poor. There is no character string data type. Programmers have to use an integer variable or an integer array for the storing of a string. Hence when two strings have to be compared, the corresponding integer arrays have to be compared word by word. Since comparisons of integer data are involved, the program has to be coded so that words of characters whose represantions are negative integer numbers can be handled correctly during the comparisons.

JOTS provides a more powerful string handling facility. Difficulties arise when the string operations have to be translated into Standard Fortran. Section 6.3.9 gives a description of how string operations are translated into Standard Fortran.

6. Implementation

This chapter describes the implementation procedures and details of building a translator for the JOTS language described in chapter 4. The implementation has been carried out on the Honeywell 6060 computer at the University of Waterloo using the YACC compiler-compiler system (see Johnson (1975)).

6.1 YACC

YACC accepts input which includes productions describing the grammar of a language, and code which is to be invoked when each production is used in a reduction. YACC then produces a parser which calls the user-supplied lexical analyser to obtain the basic tokens from the input stream.

The type of grammar accepted by YACC is LALR(1) which means the language can be parsed from left to right with a local lookahead of at most one token. Strictly speaking, LALR(1) grammars are unambiguous; that is; any sentence of the language has a unique parse tree. However, YACC accepts ambiguous grammars with appropiate disambiguating rules which are used to create parsers that are faster, easier to write and easier to understand than parsers constructed from unambiguous grammars.

An action is an arbitrary statement in a language supported by YACC. In this implementation, the system language Eh is used. Hence, the resulting output from YACC is Eh source code with the actions being functions and the parsing tables being external vectors. For a description of the Eh language, refer to Braga (1976).

6.2 Implementation procedures

Modifications have been made to the grammar of JOTS described to Chapter 4 for the input to YACC. The modifications allow certain actions to be taken at appropiate times during the parsing of a JOTS program. A few rules with actions have been added to allow the printing of appropiate error messages when syntax errors are encountered by the translator. The resulting YACC output was combined with other supporting routines and the YACC supplied main routine, thus providing a complete translator for JOTS programs.

Figure 6.1 shows the general procedure to build the JOTS translator.

```
---------------------
!    JOTS grammar   !
!    and actions    !
---------------------
          !
          !
          V
  -----------------
  !               !
  !    Y A C C    !
  !               !
  -----------------
          !
          !
          V
----------------------      -----------      --------------
! parsing tables and !      ! parser  !      ! scanner and !
!  action routines   !      !  driver !      !  supporting !
----------------------      -----------      !  routines   !
          !                      !           --------------
          !                      !                 !
          !_________________     !     ____________!
                           !     !     !
                           !     !     !
                           V     V     V
                   ----------------------
                   !  Eh compiler and   !
                   !       loader       !
                   ----------------------
                             !
                             V
                    --------------------
                    !    executable    !
                    ! JOTS translator  !
                    --------------------
```

Figure 6.1 General procedure to build the JOTS translator

6.3 Implementation techniques

The following sections outline the data structures and algorithms used in the JOTS translator.

6.3.1 Symbol table

The symbol table uses hash tables, symbol descriptors, array descriptors, initialization structures and I/O format structures. Each of these and relations among them are described below.

Hashing

Hash coding is the method used for searching the symbol table and for keeping unique Fortran names for the name mapping. Conflicts in the hashing are resolved by linking symbols having the same hash index to a "hash bucket". The data structure used for a name to be hashed consists of one word for the _hash link_, followed by words containing the name. This will be called a _hash structure_ in later sections. Two hash tables are used, one for the JOTS symbol names and one for Fortran names; each has 127 buckets. They are called the _symbol hash table_ and _Fortran hash table_, respectively. The hashing routine accepts the address of a hash table as one of its arguments and is used by both tables.

Figure 6.2 shows the relationship among the hash table and the hash structures.

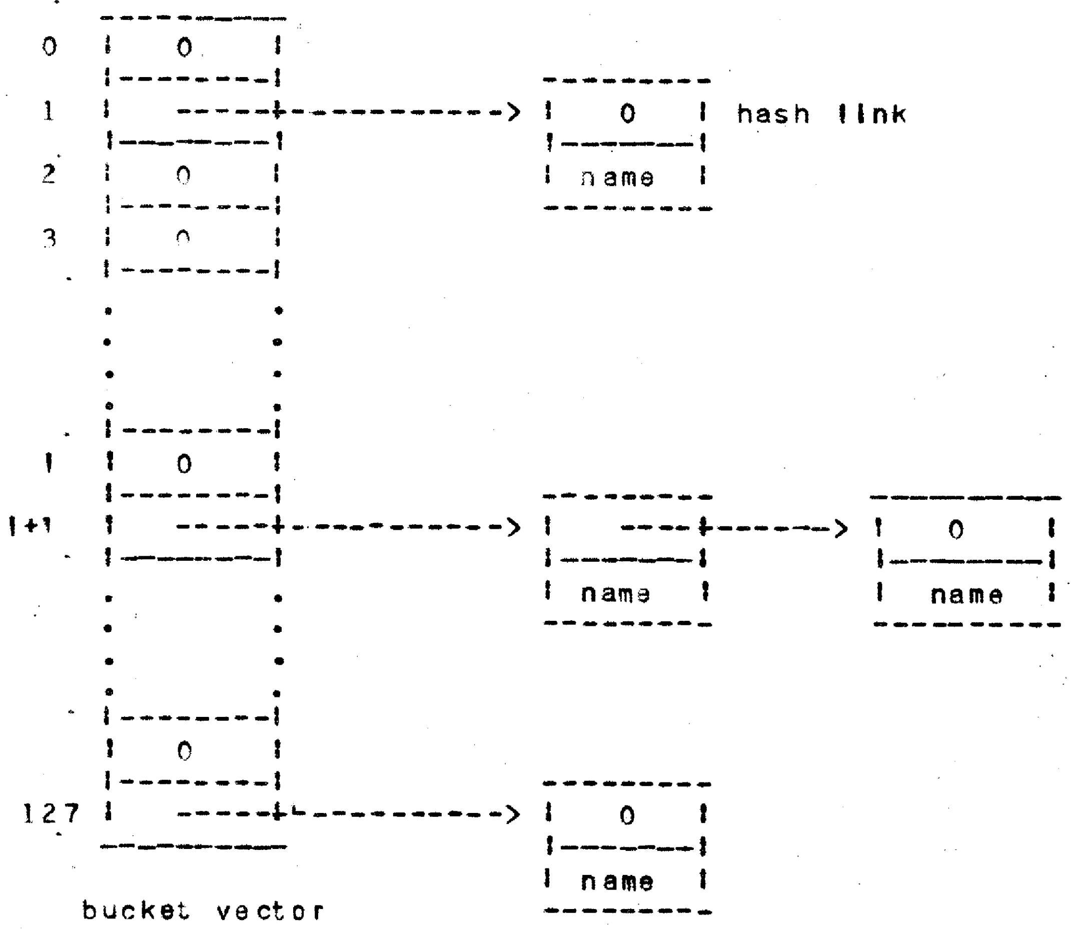


Figure 6.2 chained hash table

Symbol descriptor

A symbol descriptor is the main structure for storing information about a symbol. When a symbol is defined, a new symbol descriptor is created for it.

A symbol descriptor consists of a fixed number of words followed by a variable length name field. Figure 6.3 shows the data structure of the symbol descriptor. Word 0 of the descriptor contains the vector size of the descriptor which is used when the space for the descriptor is to be released. Word 1 of the descriptor, _init_, is a pointer to a structure which contains either initial values for the variable or an I/O format. In the descriptor, the hash link for the Fortran name, _flink_, and the Fortran name form a hash structure for the Fortran hash table as shown in Fig 6.2. Similarly, the hash link for symbol name, _nlink_, and the symbol name form a hash structure for the symbol hash table. Word 7 of the descriptor, _desc_, is a pointer to the array descriptor if the class of the symbol is array. Word 8 of the descriptor, _dlink_, points to the symbol descriptor of the next symbol declared in the source program. During emission of code for declarations, the symbol descriptors are scanned in the order that they are linked by dlink.

Table 6.1 gives the possible types and possible classes of a symbol. The valid combinations of type and class are given in Table 6.2.

When the type of a symbol desciptor is manifest, words 2 through 4 of the descriptor have a special use which is described in section 6.3.10.

```
 word
         ------------
  0      !   size   !   vector size of this descriptor
         !----------!
  1      !   init   !   pointer for initial values
         !----------!
  2      !  flink   !   hash link for Fortran name
         !----------!
  3      !          !   Fortran name (2 words)
         !----------!
  4      !          !
         !----------!
  5      !   type   !   type of the symbol
         !----------!
  6      !  class   !   class of the symbol
         !----------!
  7      !   desc   !   pointer to array descriptor
         !----------!
  8      !  dlink   !   link to next symbol declared
         !----------!
  9      !  nlink   !   hash link for symbol name
         !----------!
 10      !          !
         .          .
         .          .   symbol name
         .          .
         !          !
  n      !          !
         ------------
```

Figure 6.3 symbol descriptor

	type	class
0		simple
1	longreal	array
2	real	subprogram
3	integer	entry name
4	logical	adjustable dimension variable
5	string	predefined manifest
6	manifest constant	read format
7	label	write format
8	subroutine	print format
9	format	

Table 6.1 type and class of symbol

	CLASS								
TYPE	0	1	2	3	4	5	6	7	8
1	X	X	X	X					
2	X	X	X	X					
3	X	X	X	X	X				
4	X	X	X	X					
5	X	X							
6	X					X			
7	X								
8			X	X					
9							X	X	X

Table 6.2 Valid combinations of type and class

Array descriptor

An array descriptor contains the number of dimensions, the bound type and the corresponding number of bound descriptors for an array variable. If the array descriptor is for an automatic array formal parameter, the bound type is auto; otherwise, it is range. Due to implementation restrictions, an array can have at most three dimensions (the same restriction as ANSI Fortran). Array descriptors are dynemically allocated in a vector. Figure 6.4 shows the data structure of an array descriptor.

An array descriptor contains up to three bound descriptors depending on the number of dimensions of the array variable. The nth bound descriptor corresponds to the nth dimension. A bound descriptor has six words, the first three of which contain the offset and are called the offset descriptor, the remaining three words contain the dimension and are called the dimension descriptor. Examples for offsets and dimensions of array bound declarators are given in table 6.3.

Offsets and dimensions are stored in string form. If the bound type is auto, then each bound descriptor contains the hash structures for the offset variable name and the dimension variable name; otherwise each bound descriptor has the following format:

> The first word of the offset descriptor contains the sign of the offset value and the next two words contain the string value of the absolute offset (ABS(-lower bound + 1)); for the dimension descriptor, if the first word contains a zero, the next two words contain the numeric dimension (upper bound - lower bound + 1) in the string form, otherwise the first word is a pointer to the adjustable dimension variable.

An array descriptor is shared by the array variables declared in the same array declaration statement. However each automatic array formal parameter has its own array descriptor.

bound declarator	bound type	offset	dimension
[1:4]	range		4
[5]	range		5
[-2:10]	range	+3	13
[10:100]	range	-9	111
[N]	range		N
[*]	auto	-NOFFZ	NDIMZ

Table 6.3 Examples for offsets and dimensions of bound declarator of z.

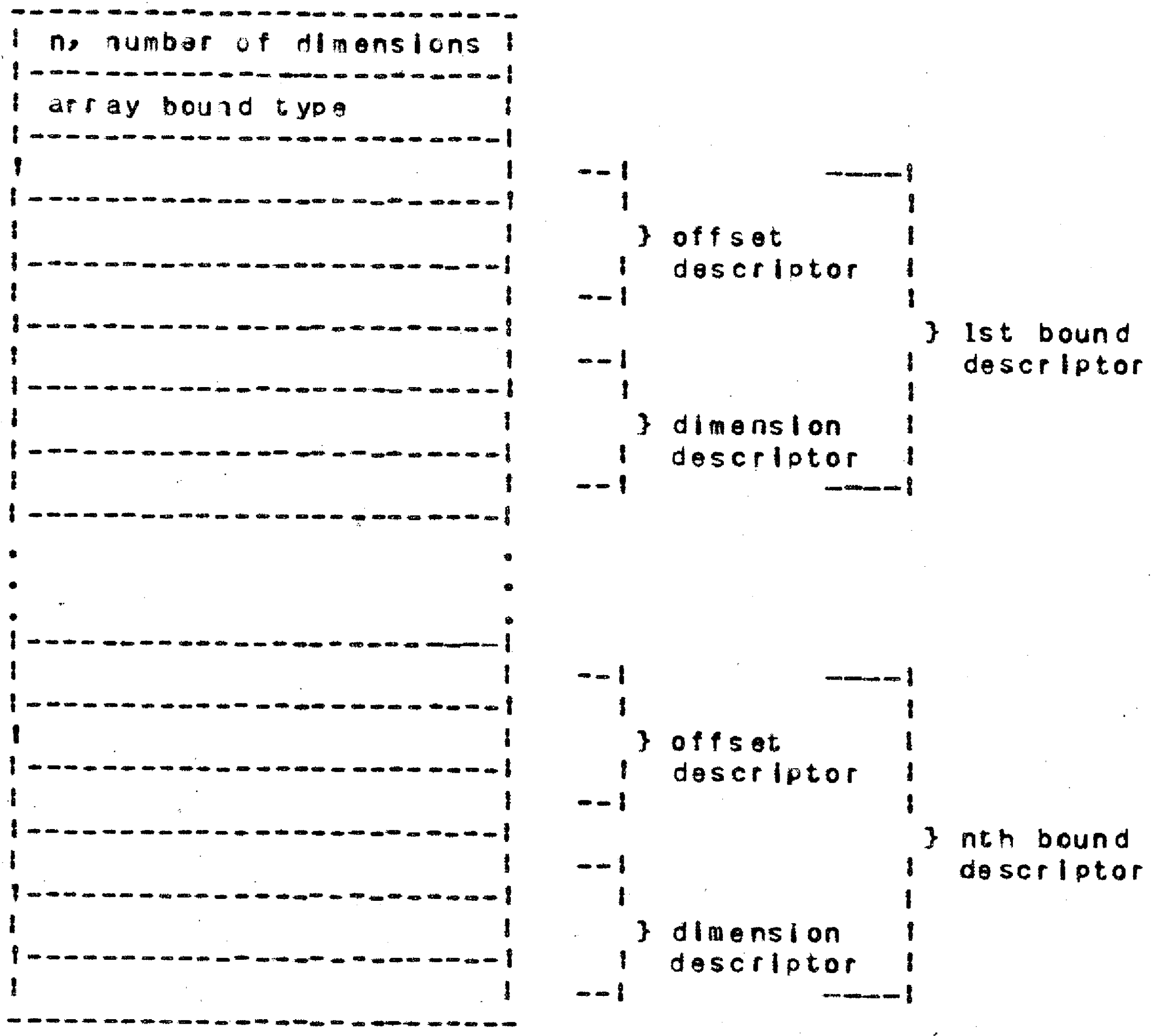


Figure 6.4 Array descriptor

Initialization storage structure

Initial values for an array can be specified in the source language as a list of initial items each of which can be a constant, or a repetitive group. A *repetitive group* is a list of initial items, separated by commas and enclosed in parentheses preceded by a repetitive count. A vector, called *gparea* is used to store the representations of initialization structures. During compilation of initial values, a constant, which is stored as a bundled structure [section 6.3.5], has its bundled address entered into the gparea. A repetitive group is represented as a list of elements into the same data area using the following order: the negative value of the repetitive count, the group items and then a zero. Initialization for an array variable is emitted as a DATA statement. If a repetitive group with a repeat count of n has only one constant item c, it is emitted as a repetitive item (i.e. n*c), otherwise the items in the group are emitted repeatedly n times. A recursive B function is used to do the above since outer groups have to be stacked during the code emission.

Figure 6.5 gives an example for the storage initialization structure.

```
                                       .         .
                                       .         .
pointed to by                          !---------!
the "init" field  ---->                !   -2    !
of the symbol                          !---------!
  descriptor                           !     ----+----> "1"
    of z                               !---------!
                                       !   -3    !
                                       !---------!
                                       !     ----+----> "2"
                                       !---------!
                                       !   -2    !
                                       !---------!
                                       !     ----+----> "5"
                                       !---------!
                                       !    0    !
                                       !---------!
                                       !    0    !
                                       !---------!
                                       !     ----+----> "-40"
                                       !---------!
                                       !    0    !
                                       !---------!
                                       !    0    !
                                       !---------!
                                       .         .
                                       .         .
                                         gparea
```

Initialization structure for the declaration :
INTEGER ARRAY[16] z = (2(1, 3(2, 2(5)), -40))

Figure 6.5 Example of an initialization structure

I/O format storage structure

An I/O format is a list of format items, each of which can be a field descriptor, or a repetitive group. A repetitive group is a list of format items separated by commas, and enclosed in parentheses preceded by a repetitive count. Representations for an I/O format are entered into the vector gparea. During the compilation of a format, a field descriptor is mapped into the Fortran field descriptor (e.g. the E format E(15,7,2) --> 2PE15.7). The string address of the mapped descriptor is enter into gparea. A repetitive group is represented in the same data area by the the following: the negative value of the repetitive count, the group nesting number which is initialized to one, the representations of the group items, and then a zero delimiter.

A recursive function scans through this structure and assigns group nesting numbers to each group. An inner most group has a nesting number of one. A group containing other groups has a nesting number one greater than the highest nesting number of its inner groups.

When a FORMAT statement is to be emitted, the group nesting number indicates whether the whole group can be emitted as a Fortran repetition group or the group has to be copied the number of times indicated by the group repeat count. A recursive function is used to do the format emission since addresses of outer groups have to be stacked during the code emission. This solves the restriction imposed by ANSI Fortran for which only two nestings of group repetition are allowed in a FORMAT statement.

An example of the representations of I/O format is given in Figure 6.6.

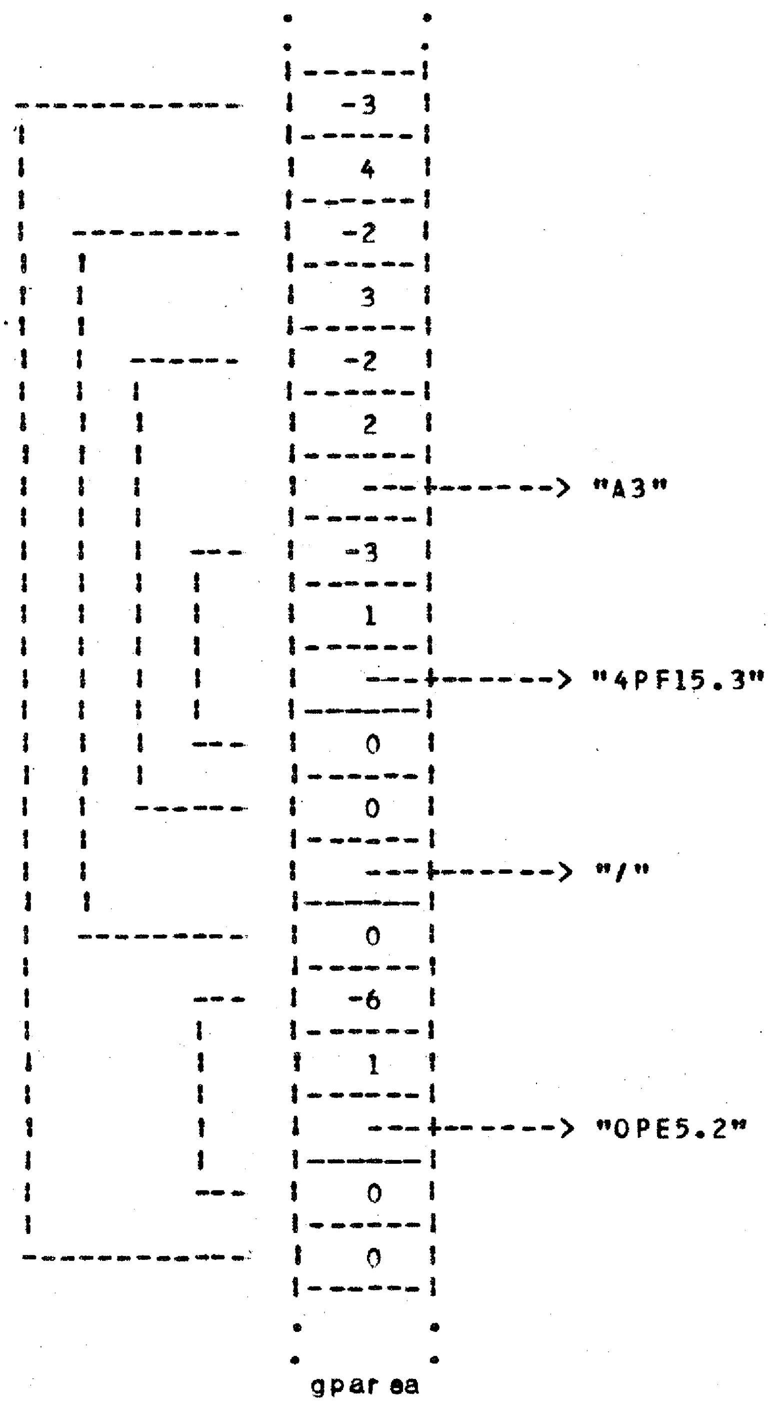


Figure 6.6 Internal representation for the I/O format :
(3(2(2(A(3), 3(F(15,3,4)))), skip), 6(E(5,2)))

6.3.2 Name mapping and generation

Generating a "readable" Fortran name

Name mapping begins with generating a legal Fortran name from the source identifier by removing appropriate characters. Underscore characters "_" are stripped from the source identifier resulting in a _stripped name_. If the length of the stripped name is five or less, it is used as the _generated name_. Otherwise five characters are chosen as follows:

> The first three and the last characters of the generated name are chosen to be the same as in the stripped name. If there is at least one underscore character "_" in the source identifier and there are at least two characters after the last underscore, the character immediately after the last underscore is used for the fourth character of the generated name. Otherwise, the character at the (length of stripped name/2 + 3)th position of the stripped name is used.

Names generated by the above algorithm are generally a pleasing short form of the source identifier.

Resolution of conflicts

The mapping algorithm gives a many-to-one mapping of source names to Fortran names. The second part of the name mapping routine resolves the conflicts, if any, for generated names. A routine NEXT_ALPHANUM(c), which returns the next alphanumeric character (according to the circular sequence "0, 1, ..., 9, A, B, C, ..., Z") of argument c, is used in the algorithm. A <u>starting symbol</u>, which is initialized to "0" when compilation of a program starts, is also used. The resulting value of the starting symbol in a conflict resolution process is kept so that in the conflict resolution of another identifier, it is used as the new starting symbol. This helps to resolve the conflicts more efficiently when several identifiers are mapped into the same name. This is especially useful for generation of temporary variables (see next section).

The generated name is denoted by g. Whenever there is a conflict, it is resolved by modifying g in the following way.

> Let c be the last character of g. Assign NEXT_ALPHANUM(starting symbol) to d and append d to g. New names are generated from g for successive trials of the conflict resolution by varying c as the major varying character and d as the minor. c or d is varied according to the alphanumeric sequence using NEXT_ALPHANUM. When all 36 x 36 combination of c and d have been tried, c and d are removed from g and the whole process is repeated.

The starting symbol is set to the last character of the resulting name.

Temporary variables for subscripts

The starting symbol for resolving name mapping conflicts is set to "0" first. When a temporary integer variable is wanted, the seed "KTEMP" is passed to the name mapping routine. By the algorithm of the name mapping routine, "KTEMP", "KTEMP1", "KTEMP2", ... would most likely be the sucessively generated temporary names.

Generated variables for subscript expressions in a statement are reused, if necessary, in later statements. This means that when a temporary integer variable for a subscript is wanted, a new name will be generated only if there are no available temporary variables.

Temporary variables for formal array parameters

The seed "NDIM" and "NOFF" are used for generating dimension variables and offset variables respectively. To make the generated variables more readable, the first character of the corresponding array name is appended as the fifth character of the seed before the seed is passed to the name mapping routine. The starting symbol for the name mapping routine is reset to "0" before the processing of an automatic array formal parameter to obtain more readable names for the temporary variables.

For example, the generated parameters for the automatic array formal parameter z[*, *], mtx[*] may be noffz, ndimz, noffz2, ndimz2, noffm, ndimm, mtx.

Notice that "may be" is used in the last statement because a previous identifier might have been mapped into "noffz" or "ndimz2" etc.

6.3.3 Declarations

Object Fortran code for the subprogram heading and all declarations in a program unit has to be emitted after all source declarations have been parsed and all name generation done. As the source declaration statements are parsed, the symbol table is built and only subprogram identifiers are entered into the Fortran name hashing table. After all declarations have been parsed, the symbol table is scanned, and every identifier other than subprogram names is mapped into a Fortran name which is then entered into the Fortran name hashing table. Emission of the Fortran code for the subprogram heading and declarations is done at the same time.

Initializations for arrays and variables are emitted as DATA statements using the initialization data structures after all declarationa have been emitted.

6.3.4 Control structures

The technique involved in the translations of the control statements into Fortran code is quite straight forward. Table 6.4 gives the translations of JOTS control statements into Fortran code.

```
JOTS control           !     Translation into Fortran
  statement            !
-----------------------+----------------------------------------
IF <c>                 !       IF                     (.NOT.
THEN                   !     .   <c'>                 ) GO TO 1001
   <s>                 !C      ..THEN
                       !            <s'>
                       ! 1001 CONTINUE
-----------------------+----------------------------------------
IF <c>                 !       IF                     (.NOT.
THEN                   !            <c'>              ) GO TO 1001
   <s>                 !C      ..THEN
ELSE                   !            <s'>
   <t>                 !                              GO TO 1002
                       ! 1001 CONTINUE
                       !C      ..ELSE
                       !            <t'>
                       ! 1002 CONTINUE
-----------------------+----------------------------------------
DO                     !C
   <s>                 ! 1001 CONTINUE
   WHILE <c>           !C      DO..
   <t>                 !           <s'>
                       !C      ...WHILE
                       !                              IF (.NOT.
                       !              <c'>            ) GO TO 1002
                       !           <t'>
                       !       GO TO 1001
                       ! 1002 CONTINUE
----------------------------------------------------------------
```

where <c> is a JOTS logical expression with translation
    <c'>
  <s> and <t> are JOTS statements with translations
    <s'> and <t'> respectively.

Table 6.4 Translations of JOTS control statements into Fortran

A global variable is used to hold the value of the current statement number used for the translations of control statements. When a new statement number is wanted, the global variable is incremented.

In the translated Fortran code, a forward branch to skip a section of code can be done by first pushing a new statement number onto a stack, then emitting a GOTO statement with this statement number and after the section of code has been emitted, the stack is popped and a CONTINUE statement is emitted with the popped value as the label. A backward branch in the translation can be implemented similarily using the same stack.

A branch label in a JOTS program is stored in a symbol descriptor. When a program unit has been compiled, a search in the symbol table is done to check for those unresolved GOTO branchings.

6.3.5 Bundling

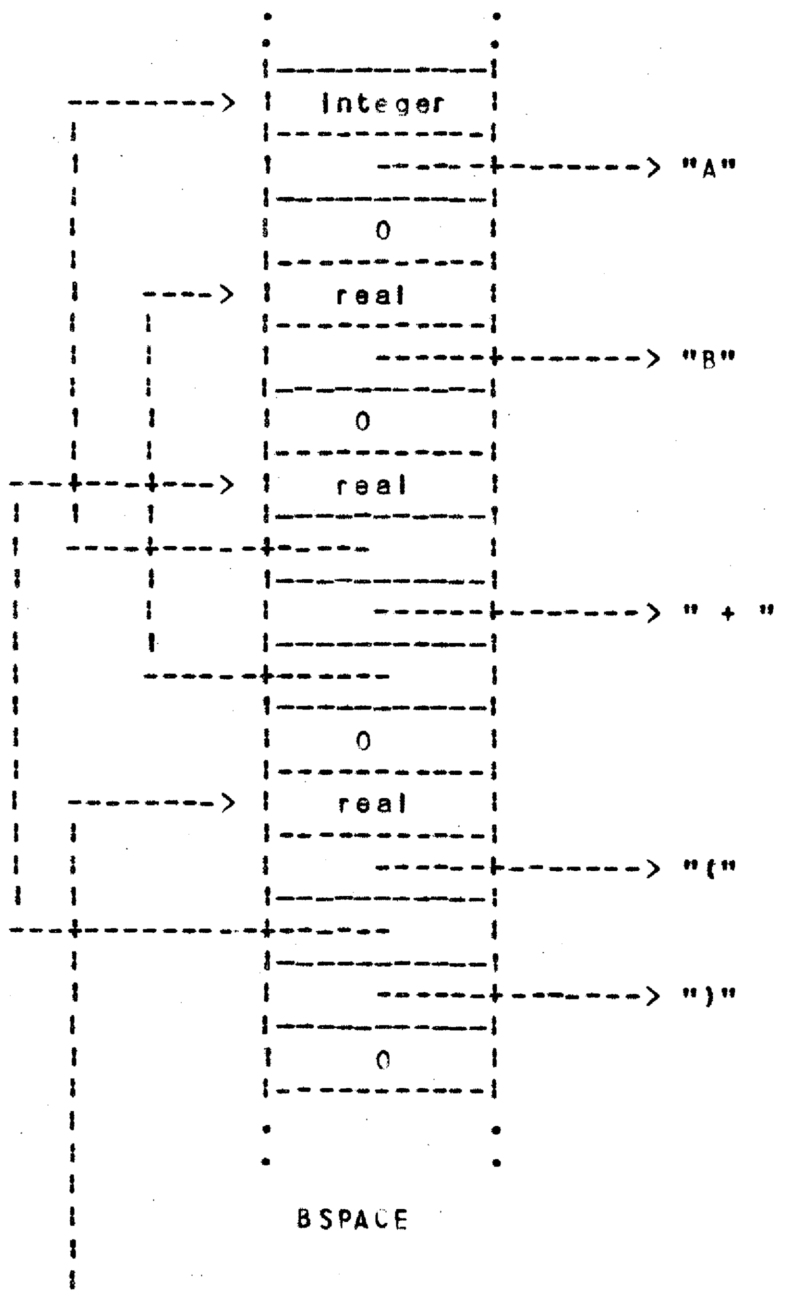
Bundling, which is derived from a feature of the same name in the compiler-compiler TMG (see McIlroy (1973)), is a technique heavily used in the implementation of the JOTS translator. Type checking, type promotions, initial values code emission and more important - the indivisible emission of a statement are implemented using this technique.

Bundling is a technique for collecting together various character strings so that they can be output at the same later time. Bundles are implemented as arrays of pointers, terminated by a zero pointer. Each pointer either points to a bundle or to a character string. There is an array, called bspace which contains all the bundles. The implementation trick is to check the various values of the pointer in a bundle to determine if it is a pointer to a string or a pointer to another bundle (points to some location in bspace). Figure 6.7 gives an example of bundling.

In order that type checking and type promotions can be done easily, the first word of a bundle is always reserved to indicate the type of the expression stored in the bundle.

Bundles are allocated sequentially in bspace and are cleared by resetting the bundle allocation pointer.

Emitting a bundle structure can be easily achieved by using a recursive procedure B_EMIT. B_EMIT accepts the address of a bundle as parameter. It skips the first word which indicates the type, and scans through the pointers one by one until the zero pointer is encountered. If a pointer p points to some location in bspace, a recursive call for B_EMIT using p as parameter is made, otherwise p is treated as a string address and emitted.



Bundle structure for (A + B)

Figure 6.7 Example of bundling.

6.3.6 Expressions

The bundling technique (section 6.3.5) helps to simplify the implementation of type checking and type promotions.

When an expression is being parsed, type checking and type promotions are performed. The translator selects the appropriate function names for built-in functions according to the types of the arguments. In an arithmetic type promotion for two operands x and y for which x has a high precedence type, if y is a constant then y is emitted as a constant that has the same type as x; otherwise if y is of integer type, it is converted to real by emitting the Fortran intrinsic function "SGNL".

For example, if dx is longreal, x is real, and i is integer then

```
        "dx*3"       is emitted as "dx*3D0",
        "x/dx"       is emitted as "x/dx",
        "2 + i/dx"   is emitted as "2D0 + SGNL(i)/dx",
        "x*35"       is emitted as "x*35.",
    and "dx*35E10"   is emitted as "dx*35D10".
```

Bundling is used to link up the names and operators for the expression in the process. If required, expressions are bundled together to form part of a statement before they are emitted. For example, all arguments of a subprogram call are bundled before they are emitted.

JOTS only restricts a subscript to be an integer expression. The translator generates a temporary integer variable to substitute a non-standard subscript expression. Code is emitted for the assignment of the non-standard subscript to the temporary variable before the actual statement containing the subscript is emitted. A semantic routine is specially written to check if a given subscript expression is standard.

A two pass method has to be used in the arithmetic type promotions for arguments of built-in functions MIN and MAX. In the first pass over the expression, every argument of the built-in function is bundled and stacked, and the highest precedence type is recorded. In the second pass, the bundled arguments are popped and appropiate conversion is performed, if needed.

6.3.7 Code emission

The object Fortran code emitted from the translator is formatted with appropriate indentations and comments. The code emission routines have to handle card boundaries, statement continuations, and indentations.

Four basic routines are responsible for the actual emission of code:

- emitting code for statements
- emitting a Fortran statement label
- emitting a comment with formatting
- emitting a comment without formatting

Routines, such as emitting a number, emitting a CONTINUE statement, emitting a bundled structure etc., are defined using the four basic routines.

6.3.8 Comments

Comments in JOTS programs have to be transcribed to object Fortran code. A Fortran statement is emitted only after the entire statement is ready. This makes it possible to transcribe comments as they are scanned by the translator.

6.3.9 Strings

Strings are implemented as integer arrays in Fortran.

A string(n) item is a string of length n.

The predefined manifest constant "BYTES_PER_WORD" contains the number of bytes in a word of the machine for which the Fortran output is to be compiled and executed. This constant can be overriden in a JOTS program.

Define words (n) as (n - 1)/BYTES_PER_WORD + 1.

A string(n) variable in JOTS is mapped into a one-dimensional integer array of size words(n) in Fortran. A m-dimensional string(n) array in JOTS is mapped into a (m+1) dimensional integer array which size of the first dimension is words(n). Since the maximum number of dimensions for arrays in ANSI Fortran is 3, string arrays in JOTS require m <= 2. BYTES_PER_WORD characters are packed into each array element. In string initializations and string assignments, unused bytes are filled with blanks.

For example, if BYTES_PER_WORD is 4, the declaration

```
 string(9) m1 = 'JANUARY'
```

is translated as

```
 INTEGER M1(3)
 DATA M1(1),M1(2),M1(3)/4HJANU,4HARY ,4H    /
```

String assignments

A Fortran subroutine SASGN has been written to handle string assignments. A string assignment s1 := s2 in JOTS is translated as

```
CALL SASGN(s1, l1, s2, l2)
```

where l1 is the value words(string size of s1)

and l2 is the value words(string size of s2)

When l1 > l2, s2 is copied to the first l2 words of s1 and the next (l1 - l2) words of s2 are filled with blanks.

String comparisons

Fortran logical functions STREQ, STRGT, etc. have been written for string comparison operations '=', '>', etc., respectively. The arguments for these functions are similar to those in SASGN. The string comparison, s1 > s2 where s1 is a string(m) variable and s2 is a string(n) variable is translated as

```
  STRGT(s1, l1, s2, l2)
      where l1 is the value words(m)
        and l2 is the value words(n).
```

Since s1 and s2 are passed as integer arrays, the comparison routines have to handle the cases where the representations of the compared items are negative.

String input/output

A string(n) variable, s appearing in an input/output list is translated as

```
   (s(ktemp), ktemp = 1, w)
      where w is the value words(n).
```

Similarly, a string(n) array element s[i] is translated as

(s(ktemp, i), ktemp = 1, w)
  where w is the value words(n).

6.3.10 Manifest constants

Word 2 to word 4 of the symbol descriptor of a manifest constant is regarded as a unit called _manifest descriptor_. The usage of each word in the descriptor is as follows:

First word _mlink_ - link to other manifest descriptor during expansion.

Second word _moffset_ - offset used during expansion

Third word _maddr_ - string address of manifest definition

When a manifest constant declaration is processed, a vector is allocated to store the definition of the manifest constant. A symbol descriptor is also allocated to the constant. The values of moffset and mlink are initially zero.

A manifest constant becomes _active_ if it is referenced in a program unit either directly or indirectly through the expansions of other manifest constants. There may be more than one manifest constant active at one time. When a manifest constant is active, its moffset is greater than zero. This fact is used to detect recursive manifest expansions which are illegal since they never terminate.

The manifest descriptors of all active manifest constants are linked in a stack called _mlist_. When the scanner invokes _getchar_ to obtain the next character, getchar first checks if mlist is empty. If it is, the next character in the input stream is returned; otherwise the corresponding character from the manifest definition of the first manifest decriptor in mlist is returned. Moffset of the manifest descriptor indicates the position of the character in the definition to be transmitted. When the last character of a manifest definition is reached, the manifest descriptor is deleted from mlist and its moffset is reset to zero to indicate that the constant is _inactive_.

7. Conclusion

Problems of portability can be divided into two categories: (1) writing code so that it can be compiled and executed on each target machine, and (2) writing code that will produce acceptable results on each target machine regardless of mathematical properties of its representations and arithmetic. JOTS is one abstraction of many Fortran machines. JOTS solves the first problem by emitting portable Fortran and by being a complete compiler preventing accidental inclusion of non-standard Fortran. JOTS also assists the programmer in solving the second problem.

In addition, the Fortran output is attractive and highly readable. This is useful for software distribution (such as the IMSL mathematical software library) for which the Fortran output can be used as a basis for code maintenance by the users.

JOTS relieves deficiencies of Fortran by providing richer control structure and precise semantics defined by BNF. Irrational restrictions of Standard Fortran (e.g. those in subscript expressions, format statements, and data statements) are removed.

Many problems are involved in using Fortran as the target language for the compiler. These difficulties arise mainly because of the restrictions in Standard Fortran. However, it is a reasonable choice because Fortran is widely available and there are highly developed subroutine libraries in Fortran.

The use of a compiler-compiler is a convenient method for software development. As a result, the JOTS language is extensible, and can be enhanced by changing or adding production rules with the corresponding actions. The translator is portable since it is implemented in the portable systems implementation language Eh.

JOTS demonstrates that it is possible to translate programs written in a phrase-structured language into portable and highly readable Fortran. It is a useful tool for writing portable software.

REFERENCES

Ansi Standard Fortran, 1966
USA standard Institute, USAS. X3.9-1966.

Beyer, T., 1974
FLECS User's manual University of Waterloo Edition, Sept 10, 1975.

Braga, R., 1976
Description of the programming language Eh, unpublished manuscript, 1976

Brent, R.P., 1973
Algorithms for Minimization without Derivations, Prentice Hall, 1973.

Gales, P.J., 1975
Structured Fortran with no preprocessors. Sigplan Notice 10, 10(Nov., 1975), pp.17-20

Gardner, M., 1970
"The fantastic combinations of John Conway's new solitaire game 'life'", Sciantic American, Oct., 1970, pp.120-123.

Johnson, S.C., 1975
YACC User's Manual, Bell Telephone Laboratories, University of Waterloo Edition.

Kernighan, B.W., 1975
A preprocessor for a rational Fortran. Software-practice and Experience 5, 4(Oct-Dec 1975), pp. 395-406.

Malcolm, M.A. and Rogers, L.D. (1974)
Workshop on Fortran preprocessor for Numerical Software. SIGNUM, 1974.

Malcolm, M.A. and Sager, G.R. (1976)
The real-time/minicomputer Laboratory, Department of Computer Science, University of Waterloo, Research Report CS-76-11, February 1976.

McIlroy, M.D., 1973
A Manual for the TMG Compiler-writing Language, 1973.

Meissner, L.P. (1975)
On Extending Fortran Control Structures to Facilitate Structured Programming, SIGPLAN NOTICE 10, 9(Sept., 1975), pp. 19-30.

REFERENCES

Ryder, B.G., 1974
The PFORT verifier. _Software-Practice and Experience_ 4, 4(Oct-Dec 1974), pp.359-377.

APPENDIX A

```
<program>          ::= <program unit list> .
<program unit list>  ::= <program unit>
                    ! <program unit list> ; <program unit>
<program unit> ::= <main program>
                    ! <subpgm>
<main program> ::= MAIN ; <program body> EXIT
<subpgm>           ::= <fcn>
                    ! <subr>
<fcn>              ::= <fcn head> <program body> RETURN
                         ( <returned Γ expr> )
<subr>             ::= <subr head> <program body> RETURN
<fcn head>         ::= <basic type> FUNCTION <entry list> ;
<subr head>        ::= SUBROUTINE <entry list> ;
                    ! SUBROUTINE <entry name> ;
<entry list>       ::= <entry name> ( <param decl list> )
<entry name>       ::= <subpgm identifier>
<returned Γ expr>      ::= <Γ expr>
<program body> ::= <declarations> <stmt list>
<declarations> ::= <decl list> ;
                    ! <empty>
<decl list>        ::= <decl list> ; <decl>
                    ! <decl>
<decl>             ::= <simple var decl>
                    ! <array decl>
                    ! <subpgm decl>
                    ! <fmt decl>
<simple type>      ::= <basic type>
                    ! STRING ( <string size> )
<basic type>       ::= INTEGER
                    ! REAL
                    ! LONGREAL
                    ! LOGICAL
<string size>  ::= <integer constant>
<simple var decl>      ::= <simple type> <s var list>
<s var list>       ::= <s var>
                    ! <s var list> , <s var>
<s var>            ::= <identifier>
                    ! <identifier> = <init value>
<array decl>       ::= <simple type> ARRAY [ <range list> ]
                         <a var list>
<a var list>       ::= <a var>
                    ! <a var list> , <a var>
<a var>            ::= <identifier>
                   ! <identifier> = <init group>
<range list>       ::= <range>
                    ! <range list> , <range>
<range>            ::= <upper range>
                    ! <lower range> : <upper range>
<lower range>      ::= <range value>
<upper range>      ::= <range value>
<range value>      ::= <integer constant>
                    ! - <integer constant>
<init group>       ::= <init value>
```

```
                     ! <repeat factor> ( <init repeat list> )
                     ! ( <init repeat list> )
<init repeat list>     ::= <init group>
                     ! <init repeat list> , <init group>
<init value>     ::= <arith constant>
                     ! - <arith constant>
                     ! <logical constant>
                     ! <string constant>
<repeat factor>        ::= <integer constant>
<arith constant>       ::= <integer constant>
                     ! <real constant>
                     ! <longreal constant>
<subpgm decl>    ::= <fcn decl>
                     ! <subr decl>
<fcn decl>       ::= EXTERNAL <basic type>
                       FUNCTION <subpgm identifier list>
<subr decl>      ::= EXTERNAL SUBROUTINE
                       <subpgm identifier list>
<subpgm identifier list>  ::= <subpgm identifier>
                     ! <subpgm identifier list> ,
                       <subpgm identifier>
<param decl list>      ::= <param decl>
                     ! <param decl list> ; <param decl>
<param decl>     ::= <param var decl>
                     ! <param array decl>
                     ! <subpgm decl>
<param var decl>       ::= <simple type> <identifier list>
<param array decl>     ::= <simple type> ARRAY [
                    <param bound list> ] <identifier list>
<param bound list>     ::= <auto bound list>
                     ! <bound list>
<auto bound list>      ::= *
                     ! <auto bound list> , *
<bound list>     ::= <bound>
                     ! <bound list> , <bound>
<bound>          ::= <range>
                     ! <adjustible dimension>
<adjustible dimension>    ::= <identifier>
<fmt decl>       ::= FORMAT ( <fmt class> ) <decl fmt list>
<fmt class>      ::= READ ! WRITE ! PRINT
<decl fmt list>        ::= <decl fmt>
                     ! <decl fmt list> , <decl fmt>
<decl fmt>       ::= <identifier> = <fmt group>
<stmt list>      ::= <stmt>
                     ! <stmt list> ; <stmt>
<stmt>           ::= <unlabelled stmt>
                     ! <label head> <unlabelled stmt>
<label head>     ::= <identifier> :
                     ! <label head> <identifier> :
<unlabelled stmt>      ::= <compound stmt>
                     ! <assignment stmt>
                     ! <call stmt>
                     ! <goto stmt>
```

```
                       ! <if stmt>
                       ! <iterative stmt>
                       ! <read stmt>
                       ! <write stmt>
                       ! <print stmt>
                       ! <rewind stmt>
                       ! <endfile stmt>
                       ! <empty>
<compound stmt>           ::= BEGIN <stmt list> END
<assignment stmt>         ::= <T0 left part> <T1 expr>
<T left part>  ::= <T var> :=
<call stmt>       ::= CALL <subpgm identifier>
                       ! CALL <subpgm identifier> ( <arg list> )
<arg list>        ::= <arg>
                       ! <arg list> , <arg>
<arg>             ::= <T expr>
                       ! <auto array arg>
<auto array arg> ::= <identifier> [ <auto indicator list> ]
<auto indicator list>       ::= *
                       ! <auto indicator list> , *
<goto stmt>       ::= GOTO <label identifier>
<label identifier>     ::= <identifier>
<if stmt>         ::= <if clause> <stmt>
                       ! <if clause> <stmt> ELSE <stmt>
<if clause>       ::= IF <logical expr> THEN
<iterative stmt>          ::= DO <stmt> WHILE <logical expr>
                          <stmt>
<read stmt>       ::= READ ( <input control> ) <io item list>
<input control>           ::= <control head>
                       ! <control head> , <input opt list>
<input opt list>          ::= <input opt>
                       ! <input opt list> , <input opt>
<input opt>       ::= <eof opt>
                       ! <err opt>
<write stmt>      ::= WRITE ( <output control> ) <output data>
<print stmt>      ::= PRINT ( <output control> ) <output data>
<output control>          ::= <control head>
                       ! <control head> , <err opt>
<output data>  ::= <io item list>
                       ! <empty>
<control head> ::= <file unit> , <fmt specifier>
<file unit>       ::= <identifier>
                       ! <integer constant>
<fmt specifier>           ::= = <fmt group>
                       ! <identifier>
                       ! *
                       ! RECORD
<eof opt>         ::= END = <label identifier>
<err opt>         ::= ERR = <label identifier>
<io item list> ::= <io item>
                       ! <io item list> , <io item>
<io item>         ::= <identifier>
                       ! <io array>
```

```
<io array>       ::= <identifier> [ <range spec list> ]
<range spec list>    ::= <range spec>
                   ! <range spec list> , <range spec>
<range spec>     ::= <integer expr>
                   ! <integer expr> : <integer expr>
                   ! *
<rewind stmt>    ::= REWIND <file unit>
<endfile stmt>   ::= ENDFILE <file unit>
<T var>          ::= <identifier>
                   ! <subpgm identifier> ( <arg list> )
                   ! <T array designator>
<T array designator> ::= <identifier> [ <subscript list> ]
<subscript list>     ::= <subscript>
                   ! <subscript list> , <subscript>
<subscript>      ::= <integer expr>
<T0 expr>        ::= <T0 var>
                   ! <built-in T0 fcn>
                   ! <T0 constant>
                   ! - <T0 expr>
                   ! + <T0 expr>
                   ! ( <T0 expr> )
                   ! <T1 expr> <arith op> <T2 expr>
<arith op>       ::= **  !  %  !  *  !  /  !  +  !  -
<string expr>    ::= <string var>
                   ! <string constant>
<logical expr> ::= <logical element>
                   ! <T1 expr> <relational op> <T2 expr>
                   ! NOT <logical expr>
                   ! <logical expr> AND <logical expr>
                   ! <logical expr> OR <logical expr>
                   ! ( <logical expr> )
<logical element>    ::= <logical var>
                   ! <logical constant>
<logical constant>   ::= TRUE
                   ! FALSE
<relational op>      ::= > ! < ! >= ! <=  ! = ! ~=
<built-in T fcn>     ::= <one-arg T fcn>
                   ! <many-arg T fcn>
<one-arg T0 fcn>     ::= <one-arg fcn name> ( <T1 expr> )
<one-arg fcn name>   ::= SHORT ! FLOAT ! LONG
                   ! TRUNCATE ! ROUND ! FLOOR ! CEILING
                   ! EXP ! LOG ! LOG10 ! SIN ! COS ! ATAN
                   ! SIGN ! SQRT
<many-arg T fcn>     ::= MAX ( <many-arg list> )
                   ! MIN ( <many-arg list> )
<many-arg list>      ::= <T1 expr> , <T2 expr>
                   ! <many-arg list> , <T3 expr>
<fmt group>      ::= <edit desc>
                   ! SKIP ( <integer constant> )
                   ! SKIP
                   ! X ( <integer constant> )
                   ! PAGE
                   ! <string constant>
```

```
                      | <integer constant> ( <fmt repeat list> )
                      | ( <fmt repeat list> )
<fmt repeat list>       ::= <fmt group>
                      | <fmt repeat list> , <fmt group>
<edit desc>       ::= <fractioral field> ( <fractional desc> )
                      | I ( <field width> )
                      | A ( <field width> )
                      | L ( <field width> )
<fractional field>    ::= D | E | F | G
<fractional desc> ::= <field width> , <fractional digit> ,
                        <scale factor>
                      | <field width> , <fractional digit>
<field width>   ::= <integer constant>
<fractional digit>    ::= <integer constant>
<scale factor> ::= <integer constant>
                      | - <integer constant>
```

APPENDIX B

The ALGOL procedure zero:
(p58, Brent(1973))

```
REAL PROCEDURE zero(a, b, macheps, t, f);
VALUE a, b, macheps, t; REAL a, b, macheps;
REAL PROCEDURE f;
BEGIN
    REAL c, d, e, fa, fb, fc, tol, m, p, q, r, s;
    fa := f(a);  fb := f(b);

    int :
    c := a; fc := fa; d := e := b - a;

    ext :
    IF abs(fc) < abs(fb)
    THEN BEGIN
        a := b; b:= c; c := a;
        fa := fb; fb := fc; fc := fa
    END;
    tol := 2*macheps*abs(b) + t;
    m := 0.5*(c - b);
    IF abs(m) > tol  and fb ~= 0
    THEN BEGIN  COMMENT see IF bisection is forced;
        IF abs(e) < tol  or  abs(fa) <= abs(fb)
        THEN
            d := e := m;
        ELSE BEGIN
            s := fb/fa;
            IF a = c
            THEN BEGIN  COMMENT Linear interpolation;
                p := 2*m*s;
                q := 1 - s
            END
            ELSE BEGIN  COMMENT Inverse quadratic interpolation;
                q := fa/fc;
                r := fb/fc;
                p := s*(2*m*q*(q - r) - (b - a)*(r - 1));
                q := (q - 1)*(r - 1)*(s - 1)
            END;
            IF p > 0
            THEN
                q := -q
            ELSE
                p := -p;
            s := e; e := d;
            IF 2*p < 3*m*q - abs(tol*q)
                and p < abs(0.5*s*q)
            THEN
                d := p/q
            ELSE
                d := e := m
        END;
```

```
          a := b; fa := fb;
          b := b + (IF abs(d) > tol THEN d
              ELSE IF m > 0 THEN tol ELSE -tol);
          fb := f(b);
          IF fb > 0 and fc > 0  or  fb <= 0 and fc <= 0
          THEN
              GOTO int;
          GOTO ext;
      zero := b
  END zero;
```

A Fortran translation of the ALGOL procedure zero:
(Brent (1973, p.188))

```
      REAL FUNCTION ZERO(A, B, MACHEP, T, F)
      REAL A,B,MACHEP,T,F,SA,SB,C,D,E,FA,FB,FC,TOL,M,P,Q,R,S
      SA = A
      SB = B
      FA = F(SA)
      FB = F(SB)
   10 C = SA
      FC = FA
      E = SB - SA
      D = E
   20 IF (ABS(FC) .GE. ABS(FB)) GO TO 30
      SA = SB
      SB = C
      C = SA
      FA = FB
      FB = FC
      FC = FA
   30 TOL = 2.0*MACHEP*ABS(SB) + T
      M = 0.5*(C - SB)
      IF ((ABS(M) .LE. TOL) .OR. (FB .EQ. 0.0)) GO TO 140
      IF ((ABS(E) .GE. TOL) .AND. (ABS(FA) .GT. ABS(FB))) GO TO 40
      E = M
      D = E
      GO TO 100
   40 S = FB/FA
      IF (SA .NE. C) GO TO 50
      P = 2.0*M*S
      Q = 1.0 - S
      GO TO 60
   50 Q = FA/FC
      R = FB/FC
      P = S*(2.0*M*Q*(Q - R) - (SB - SA)*(R - 1.0))
      Q = (Q - 1.0)*(R - 1.0)*(S - 1.0)
   60 IF (P .LE. 0.0) GO TO 70
      Q = -Q
      GO TO 80
   70 P = -P
   80 S = E
      E = D
      IF ((2.0*P .GE. 3.0*M*Q-ABS(TOL*Q)) .OR. (P .GE. ABS(0.5*S*Q)))
     .             GO TO 90
      D = P/Q
      GO TO 100
   90 E = M
      D = E
  100 SA = SB
      FA = FB
```

```
    IF (ABS(D) .LE. TOL) GO TO 110
    SB = SB + D
    GO TO 130
110 IF (M .LE. 0.0) GO TO 120
    SB = SB + TOL
    GO TO 130
120 SB = SB - TOL
130 FB = F(SB)
    IF ((FB .GT. 0.0) .AND. (FC .GT. 0.0)) GO TO 10
    IF ((FB .LE. 0.0) .AND. (FC .LE. 0.0)) GO TO 10
    GO TO 20
140 ZERO = SB
    RETURN
    END
```

The JOTS procedure zero is given below:

```
!$d
real function zeroin(real a, b, t;
                     external real function f);
   real c, d, e, fa, fb, fc, tolerance, m, p, q, r, s;
   fa := f(a);  fb := f(b);

   int :
   c := a; fc := fa; d := b - a;  e := d;

   ext :
   if abs(fc) < abs(fb)
   then begin
       a := b; b:= c; c := a;
       fa := fb; fb := fc; fc := fa
   end;
   tolerance := 2*machine_eps*abs(b) + t;
   m := 0.5*(c - b);
   if abs(m) > tolerance  and fb ~= 0
   then begin  ! see if bisection is forced
       if abs(e) < tolerance  or  abs(fa) <= abs(fb)
       then begin
           d := m;
           e := m
       end
       else begin
           s := fb/fa;
           if a = c
           then begin ! Linear interpolation
               p := 2*m*s;
               q := 1 - s
           end
           else begin  ! Inverse quadratic interpolation
               q := fa/fc;
               r := fb/fc;
               p := s*(2*m*q*(q-r) - (b - a)*(r-1));
               q := (q - 1)*(r - 1)*(s - 1)
           end;
           if p > 0
           then
               q := -q
           else
               p := -p;
           s := e; e := d;
           if 2*p < 3*m*q - abs(tolerance*q)
               and p < abs(0.5*s*q)
           then
               d := p/q
```

```
            else begin
                d := m;
                e := m
            end
        end;
        a := b; fa := fb;
        if abs(b) > tolerance
        then
            b := b + d
        else
            if m > 0
            then
                b := b + tolerance
            else
                b := b - tolerance;
        fb := f(b);
        if fb > 0 and fc > 0  or  fb <= 0 and fc <= 0
        then
            goto int;
        goto ext;
    end;
return(b).
```

APPENDIX B

The Fortran output of the JOTS procedure zero from the JOTS compiler:

```
C
C
C$d
C
      REAL    FUNCTION ZEROIN(A, B, T, F)
        LOGICAL STREQ, STRNE, STRGT, STRGE, STRLT, STRLE
        REAL    A, B, T
        REAL    F
        EXTERNAL F
        REAL    C, D, E, FA, FB, FC, TOLAE, M, P, Q, R, S
C
        FA = F(A)
        FB = F(B)
C
  101   CONTINUE
        C = A
        FC = FA
        D = B - A
        E = D
C
  102   CONTINUE
        IF                                          (  .NOT.
     .    (ABS(FC) .LT. ABS(FB))                     ) GO TO 5001
C        ..THEN
            A = B
            B = C
            C = A
            FA = FB
            FB = FC
            FC = FA
 5001   CONTINUE
C
        TOLAE = ((2.*0.74506E-8)*ABS(B)) + T
        M = 0.5*(C - B)
        IF                                          (  .NOT.
     .    ((ABS(M) .GT. TOLAE) .AND. (FB .NE. 0.))   ) GO TO 5002
C        ..THEN
C           ... see if bisection is forced
            IF                                      (  .NOT.
     .        ((ABS(E) .LT. TOLAE) .OR. (ABS(FA) .LE. ABS(FB)))
     .                                              ) GO TO 5003
C           ..THEN
                D = M
                E = M
                                                     GO TO 5004
 5003       CONTINUE
C           ..ELSE
                S = FB/FA
                IF                                        (  .NOT.
     .            (A .EQ. C)                              ) GO TO 5005
C                 ..THEN
C                       ... Linear interpolation
                        P = (2.*M)*S
                        Q = 1. - S
                                                            GO TO 5006
 5005           CONTINUE
C                 ..ELSE
C                       ... Inverse quadratic interpolation
                        Q = FA/FC
                        R = FB/FC
                        P = S*((((2.*M)*Q)*(Q - R)) - ((B - A)*(R - 1.)
     .                     ))
                        Q = ((Q - 1.)*(R - 1.))*(S - 1.)
 5006           CONTINUE
C
                IF                                        (  .NOT.
     .            (P .GT. 0.)                             ) GO TO 5007
C                 ..THEN
                        Q = -Q
                                                            GO TO 5008
 5007           CONTINUE
C                 ..ELSE
                        P = -P
 5008           CONTINUE
C
                S = E
                E = D
                IF                                        (  .NOT.
     .            (((2.*P) .LT. (((3.*M)*Q) - ABS(TOLAE*Q))) .AND.
     .            (P .LT. ABS((0.5*S)*Q)))                ) GO TO 5009
C                 ..THEN
                        D = P/Q
                                                            GO TO 5010
 5009           CONTINUE
C                 ..ELSE
                        D = M
                        E = M
 5010           CONTINUE
C
 5004         CONTINUE
C
              A = B
              FA = FB
              IF                                          (  .NOT.
     .          (ABS(B) .GT. TOLAE)                         ) GO TO 5011
C              ..THEN
                  B = B + D
                                                            GO TO 5012
 5011         CONTINUE
```

```
C                ..ELSE
                     IF                                          (   .NOT.
      .                (M  .GT. 0.)                              ) GO TO 5013
C                     ..THEN
                          B = B + TOLAE
                                                                    GO TO 5014
 5013                CONTINUE
C                     ..ELSE
                          B = B - TOLAE
 5014                CONTINUE
C
 5012             CONTINUE
C
                  FB = F(B)
                  IF                                             (   .NOT.
      .             (((FB .GT. 0.) .AND. (FC .GT. 0.)) .OR. ((FB .LE. 0.)
      .              .AND. (FC .LE. 0.)))                        ) GO TO 5015
C                ..THEN
                     GO TO 101
 5015             CONTINUE
C
                  GO TO 102
 5002        CONTINUE
C
             ZEROIN = B
             RETURN
           END
```

Source listing of the program "game of life" (Gardner (1970)) from JOTS:

```
*** JOTS COMPILER Version 1 Level 0 (MAY 76)***
                    TODAY IS 06/22/76     TIME 20:26:46

line        -------- input --------

   1     |!$DUMP
   2     |#death 0
   3     |#alive 1
   4     |subroutine clear(integer size;
   5     |                 integer array[*, *] sq_matrix);
   6     |   integer i, j;
   7     |   i := 0;
   8     |   do while i <= size + 1
   9     |   begin
  10     |      j := 0;
  11     |      do while j <= size + 1
  12     |      begin
  13     |         sq_matrix[i, j] := death;
  14     |         j := j + 1
  15     |      end;
  16     |      i := i + 1
  17     |   end
  18     |return;


*** SYMBOL TABLE DUMP ***

         name               fortran name    type      class


   CLEAR                    CLEAR                     entry
   SIZE                     SIZE            integer
   SQ_MATRIX                SQMRX           integer   array
   I                        I               integer
   J                        J               integer

  19     |!$~DUMP
  20     |!
  21     |subroutine input(integer size;
  22     |                 integer array[*, *] generation);
  23     |   integer x, y;
  24     |   do
  25     |      read(card_reader, =2(i(1))) x, y
  26     |      while x ~= 0
```

```
|       generation[x, y] := alive
|return;
|!
|!
|subroutine copy(integer size; integer array[*,*]
|                         to_matrix, from_matrix);
|   integer i,j;
|   i := 1;
|   do while i <= size
|   begin
|      j := 1;
|      do while j <= size
|      begin
|         to_matrix[i,j] := from_matrix[i,j];
|         j := j + 1
|      end;
|      i := i + 1
|   end
|return;
|
|
|subroutine output(integer size;
|                        integer array[*, *] generation);
|   string(1) star = '*', blank =' ';
|   string(1) array[10] buffer;
|   integer i, j;
|   i := 1;
|   do while i <= size
|   begin
|      j := 1;
|      do while j <= size
|      begin
|         if generation[i, j] = 1
|         then
|            buffer[j] := star
|         else
|            buffer[j] := blank;
|         j := j + 1
|      end;
|      print(printer, =20(a(1))) buffer[1:size];
|      i := i + 1
|   end
|return;
|
|
|!$dump
|!  game of life
|
|main;
|   integer array[0:11, 0:11] generation, next_generation;
|   integer i, j, m, number_of_generation, number_of_nbr;
|   integer board_size;
|   integer ip1, im1, jp1, jm1;
|   external subroutine input, output, clear, copy;
|
|   read(card_reader, =2(i(2))) board_size, number_of_generation;
|   call clear(board_size, generation[*,*]);
|   call clear(board_size, next_generation[*,*]);
|   call input(board_size, generation[*,*]);
|   print(printer, =('original pattern:', skip(2)));
|   call output(board_size, generation[*,*]);
|!
|   m := 1;
|   do while m <= number_of_generation
|   begin
|      i := 1;
|      do while i <= board_size
|      begin
|         ip1 := i + 1;
|         im1 := i - 1;
|         j := 1;
|         do while j <= board_size
|         begin
|            ! find neighbours of cell(i,j)
|            jp1 := j + 1;
|            jm1 := j - 1;
|            number_of_nbr :=
|               generation[im1,jm1] + generation[im1,j] +
|               generation[im1,jp1] + generation[i,jm1] +
|               generation[i,jp1] + generation[ip1,jm1] +
|               generation[ip1,j] + generation[ip1,jp1];
|            !assume death for next generation
|            next_generation[i,j] := death;
|            if (generation[i,j] = death
|               and number_of_nbr = 3)
|            then
|               next_generation[i,j] := alive
|            else
|               if generation[i,j] = alive and
|                  (number_of_nbr = 2 or number_of_nbr = 3)
|               then
|                  next_generation[i, j] := alive;
|            j := j + 1
|         end;
|         i := i + 1
|      end;
|      print(printer, =(skip(2), 'generation', i(3), ':')) m;
|      call output(board_size, next_generation[*, *]);
|      call copy(board_size, generation[*,*],
|         next_generation[*,*]);
|      m := m + 1
|   end
|exit.
```

```
*** SYMBOL TABLE DUMP ***

        name                      fortran name   type         class

GENERATION                        GENTN          integer      array
NEXT_GENERATION                   NEXGN          integer      array
I                                 I              integer
J                                 J              integer
M                                 M              integer
NUMBER_OF_GENERATION              NUMGN          integer
NUMBER_OF_NBR                     NUMNR          integer
BOARD_SIZE                        BOASE          integer
IP1                               IP1            integer
IM1                               IM1            integer
JP1                               JP1            integer
JM1                               JM1            integer
INPUT                             INPUT          subroutine   subprogram
OUTPUT                            OUTPUT         subroutine   subprogram
CLEAR                             CLEAR          subroutine   subprogram
COPY                              COPY           subroutine   subprogram
```

Object Fortran code for program "games of life" from the JOTS compiler:

```
C
C$DUMP
C
      SUBROUTINE CLEAR(SIZE, SQMRX, NOFFS, NDIMS, NOFFS2, NDIMS2)
        LOGICAL STREQ, STRNE, STRGT, STRGE, STRLT, STRLE
        INTEGER SIZE
        INTEGER SQMRX(NDIMS, NDIMS2)
        INTEGER I, J
C
        I = 0
C
 5001   CONTINUE
C        DO..
C        ...WHILE
                                                    IF (.NOT.
     .          (I .LE. (SIZE + 1))                ) GO TO 5002
            J = 0
C
 5003       CONTINUE
C            DO..
C            ...WHILE
                                                    IF (.NOT.
     .              (J .LE. (SIZE + 1))        .   ) GO TO 5004
                KTEMP = I - NOFFS
                KTEMP1 = J - NOFFS2
                SQMRX(KTEMP, KTEMP1) = 0
                J = J + 1
            GO TO 5003
 5004       CONTINUE
C
            I = I + 1
        GO TO 5001
 5002   CONTINUE
C
        RETURN
      END
C
C$~DUMP
C
C
      SUBROUTINE INPUT(SIZE, GENTN, NOFFG, NDIMG, NOFFG2, NDIMG2)
        LOGICAL STREQ, STRNE, STRGT, STRGE, STRLT, STRLE
        INTEGER SIZE
        INTEGER GENTN(NDIMG, NDIMG2)
        INTEGER X, Y
C
C
 5001   CONTINUE
C        DO..
```

```
            READ(5, 9001) X, Y
 9001       FORMAT(2(I1))
C        ...WHILE
                                                  IF (.NOT.
     .        (X .NE. 0)                          ) GO TO 5002
            KTEMP = X - NOFFG
            KTEMP1 = Y - NOFFG2
            GENTN(KTEMP, KTEMP1) = 1
         GO TO 5001
 5002    CONTINUE
C
         RETURN
       END
C
C
C
C
       SUBROUTINE COPY(SIZE, TOMRX, NOFFT, NDIMT, NOFFT2, NDIMT2, FROMX
     .   , NOFFF, NDIMF, NOFFF2, NDIMF2)
         LOGICAL STREQ, STRNE, STRGT, STRGE, STRLT, STRLE
         INTEGER SIZE
         INTEGER TOMRX(NDIMT, NDIMT2), FROMX(NDIMT, NDIMT2)
         INTEGER I, J
C
         I = 1
C
 5001    CONTINUE
C         DO..
C        ...WHILE
                                                  IF (.NOT.
     .        (I .LE. SIZE)                       ) GO TO 5002
            J = 1
C
 5003       CONTINUE
C            DO..
C           ...WHILE
                                                  IF (.NOT.
     .           (J .LE. SIZE)                    ) GO TO 5004
               KTEMP = I - NOFFT
               KTEMP1 = J - NOFFT2
               KTEMP2 = I - NOFFF
               KTEMP3 = J - NOFFF2
               TOMRX(KTEMP, KTEMP1) = FROMX(KTEMP2, KTEMP3)
               J = J + 1
            GO TO 5003
 5004       CONTINUE
C
            I = I + 1
         GO TO 5001
 5002    CONTINUE
C
         RETURN
       END
```

```
C
C
      SUBROUTINE OUTPUT(SIZE, GENTN, NOFFG, NDIMG, NOFFG2, NDIMG2)
        LOGICAL STREQ, STRNE, STRGT, STRGE, STRLT, STRLE
        INTEGER SIZE
        INTEGER GENTN(NDIMG, NDIMG2)
        INTEGER STAR(1), BLANK(1)
        INTEGER BUFER(1, 10)
        INTEGER I, J
        DATA STAR(1)/4H*   /
        DATA BLANK(1)/4H    /
C
        I = 1
C
 5001   CONTINUE
C        DO..
C        ...WHILE
                                                     IF (.NOT.
     .        (I .LE. SIZE)                          ) GO TO 5002
            J = 1
C
 5003       CONTINUE
C            DO..
C            ...WHILE
                                                     IF (.NOT.
     .            (J .LE. SIZE)                      ) GO TO 5004
                KTEMP = I - NOFFG
                KTEMP1 = J - NOFFG2
                IF                                   (  .NOT.
     .            (GENTN(KTEMP, KTEMP1) .EQ. 1)         ) GO TO 5005
C                ..THEN
                    CALL SASGN(BUFER(1, J), 1, STAR, 1)
                                                       GO TO 5006
 5005           CONTINUE
C                ..ELSE
                    CALL SASGN(BUFER(1, J), 1, BLANK, 1)
 5006           CONTINUE
C
                J = J + 1
            GO TO 5003
 5004       CONTINUE
C
            WRITE(6, 9001) ((BUFER(KTEMP1, KTEMP), KTEMP1 = 1, 1),
     .        KTEMP = 1, SIZE)
 9001       FORMAT(1H , 20(A1))
            I = I + 1
        GO TO 5001
 5002   CONTINUE
C
        RETURN
      END
```

```
C
C$DUMP
C  game of life
C
C       MAIN...
          LOGICAL STREQ, STRNE, STRGT, STRGE, STRLT, STRLE
          INTEGER GENTN(12, 12), NEXGN(12, 12)
          INTEGER I, J, M, NUMGN, NUMNR
          INTEGER BOASE
          INTEGER IP1, IM1, JP1, JM1
          EXTERNAL INPUT, OUTPUT, CLEAR, COPY
C
          READ(5, 9001) BOASE, NUMGN
 9001     FORMAT(2(I2))
          CALL CLEAR(BOASE, GENTN,  - 1, 12,  - 1, 12)
          CALL CLEAR(BOASE, NEXGN,  - 1, 12,  - 1, 12)
          CALL INPUT(BOASE, GENTN,  - 1, 12,  - 1, 12)
          WRITE(6, 9002)
 9002     FORMAT(1H ,  17Horiginal pattern:/1H0)
          CALL OUTPUT(BOASE, GENTN,  - 1, 12,  - 1, 12)
C
          M = 1
C
 5001     CONTINUE
C          DO..
C          ...WHILE
                                                         IF (.NOT.
     .           (M .LE. NUMGN)                             ) GO TO 5002
               I = 1
C
 5003          CONTINUE
C               DO..
C               ...WHILE
                                                         IF (.NOT.
     .                (I .LE. BOASE)                       ) GO TO 5004
                    IP1 = I + 1
                    IM1 = I - 1
                    J = 1
C
 5005               CONTINUE
C                    DO..
C                    ...WHILE
                                                         IF (.NOT.
     .                     (J .LE. BOASE)                  ) GO TO 5006
C                        ... find neighbours of cell(i,j)
                         JP1 = J + 1
                         JM1 = J - 1
                         NUMNR = ((((((GENTN(IM1 + 1, JM1 + 1) + GENTN(
     .                     IM1 + 1, J + 1)) + GENTN(IM1 + 1, JP1 + 1))
     .                      + GENTN(I + 1, JM1 + 1)) + GENTN(I + 1, JP1
     .                      + 1)) + GENTN(IP1 + 1, JM1 + 1)) + GENTN(
     .                     IP1 + 1, J + 1)) + GENTN(IP1 + 1, JP1 + 1)
C                        ... assume death for next generation
```

```
                        NEXGN(I + 1, J + 1) = 0
                        IF                          ( .NOT.
     .                    ((GENTN(I + 1, J + 1) .EQ. 0) .AND. (NUMNR
     .                     .EQ. 3))                    ) GO TO 5007
C                       ..THEN
                           NEXGN(I + 1, J + 1) = 1
                                                         GO TO 5008
 5007                   CONTINUE
C                       ..ELSE
                           IF                       ( .NOT.
     .                       ((GENTN(I + 1, J + 1) .EQ. 1) .AND. ((
     .                       NUMNR .EQ. 2) .OR. (NUMNR .EQ. 3)))
     .                                                ) GO TO 5009
C                          ..THEN
                              NEXGN(I + 1, J + 1) = 1
 5009                      CONTINUE
C
 5008                   CONTINUE
C
                        J = J + 1
                     GO TO 5005
 5006                CONTINUE
C
                     I = I + 1
                  GO TO 5003
 5004             CONTINUE
C
                  WRITE(6, 9003) M
 9003             FORMAT(1H /1H0, 10Hgeneration, I3, 1H:)
                  CALL OUTPUT(BOASE, NEXGN, - 1, 12, - 1, 12)
                  CALL COPY(BOASE, GENTN, - 1, 12, - 1, 12, NEXGN, - 1,
     .               12, - 1, 12)
                  M = M + 1
               GO TO 5001
 5002          CONTINUE
C
               STOP
              END
```

APPENDIX B

The following is a sample output from the "game of life".

The input data:

```
 7 8
34
43
44
45
00
```

The output from the program:
(The first column of each line contains the carriage
  control character)

```
 original pattern:
0


    *
   ***




0generation  1:


   ***
   ***
    *



0generation  2:

    *
   * *

   ***



0generation  3:

    *
    *
   * *
    *
    *
```

```
Ogeneration  4:


   ***
   * *
   ***



Ogeneration  5:

    *
   * *
  *   *
   * *
    *


Ogeneration  6:

    *
   ***
  ** **
   ***
    *


Ogeneration  7:

   ***
  *   *
  *   *
  *   *
   ***


Ogeneration  8:
    *
   ***
  * * *
 *** ***
  * * *
   ***
    *
```

APPENDIX C

Listing of pre-defined manifest constants:

pre-defined manifest	default value used for H6060
MAX_INTEGER	34359738367
MAX_REAL	1.7E38
MACHINE_EPS	0.74506E-8
BYTES_PER_WORD	4
CARD_READER	5
PRINTER	6
PUNCH	7