SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 4, 359-377 (1974)

The PFORT Verifier

B. G. RYDER

Bell Laboratories, Murray Hill, New Jersey 07974, U.S.A.

SUMMARY

The PFORT Verifier is a program which checks a FORTRAN program (i.e. a main program and a set of subprograms) for adherence to a large, carefully defined, portable subset of American National Standard FORTRAN called PFORT. Unlike many FORTRAN implementations, the Verifier diagnoses errors in interprogram-unit communication through argument lists and COMMON. The Verifier is itself written in PFORT and has been installed on a variety of computers. This paper describes the development of PFORT and the Verifier. A detailed definition of PFORT noting its differences from ANS FORTRAN is included.

KEY WORDS FORTRAN Software portability

INTRODUCTION

The PFORT Verifier is a program which checks a FORTRAN program (*i.e.* a main program and a set of subprograms) for adherence to PFORT, a portable subset of American National Standard FORTRAN.[1-3] Most FORTRAN compilers check only that individual program units are syntactically correct; the Verifier in addition checks that interprogram-unit communication, which occurs through the use of COMMON and argument lists, is consistent with the Standard. However, the Verifier is not a compiler since it has no code generator.

The Verifier provides a number of facilities useful in debugging and documentation. It produces intraprogram-unit error diagnostics, symbol tables and cross reference tables. The interprogram-unit documentation for each program unit includes its arguments, common blocks, the program units it calls and the program units which call it. A list of global common definitions is also produced. This information is especially useful in the debugging of large FORTRAN systems.

In the interests of portability, the Verifier itself is written in PFORT. To install the Verifier, two small assembly language subprograms for the packing and unpacking of character data in a machine word must be written and some parameters describing properties of the host machine must be initialized.[4]

This paper is composed of three major sections and an Appendix. The first describes the development of PFORT, a portable subset of ANS FORTRAN. The second briefly illustrates some specific situations in interprogram-unit communications which cause portability difficulties. The third section presents our experience with the Verifier. The Appendix is a complete definition of PFORT, including how it differs from ANS FORTRAN.

Received 8 April 1974

THE DEVELOPMENT OF PFORT

Computer software must be portable to ensure usage by a large diverse group of people. One method of attacking the problem of portability is to carefully define a common subset of some programming language available on a variety of computers.

In the mid-1960s a group of FORTRAN manufacturers established a definition of the FORTRAN language called American National Standard FORTRAN. The definition included restrictions which would increase the uniformity of the language on all compilers implementing the Standard. However, the Standard falls short of being a solution to the problem of portability. Firstly, it is difficult to interpret, making clarifications necessary.[1] [2] Secondly, *there is no automatic method of forcing adherence to it.* Manual referral to the Standard to check the conformance of code is at best exceedingly difficult. Therefore, we chose to carefully and clearly define a large portable subset of ANS FORTRAN (i.e. PFORT) and write a program (i.e. the Verifier) to check conformance with this portable subset.

PFORT is a large subset of ANS FORTRAN which is incorporated in the major FORTRAN dialects.[5] PFORT differs from ANS FORTRAN in that additional restrictions have been placed on the ordering of statements in a program unit and on the ordering of data types in common. Constructions directly related to machine specifications, such as the number of characters in a word, are prohibited. Conflicting implementation decisions, arising from lack of specification of implementation in the Standard, are noted as possible sources of portability difficulty (e.g. argument passing in function or subroutine calls).

Merely describing the differences between PFORT and ANS FORTRAN does not define PFORT, as the Standard is *not* a clear reference document for FORTRAN. Thus, we have presented a complete syntactic description of PFORT in the Appendix. PFORT is a large subset of ANS FORTRAN; therefore, this exercise practically serves as an explanation of ANS FORTRAN as well.

The syntax is based upon McIlroy's ANS FORTRAN syntax[6] and Hall's modifications.[5] PFORT has been validated through usage in the ALTRAN system[7] as well as by painstaking reading of FORTRAN manuals and the Standard.

The Verifier checks a complete program for compliance with most, but not all, of the rules in PFORT. Rules which involve extensive flow analysis or storage allocation analysis of a program are not checked; these were judged to be less applicable to our portability goal than other rules, and thus were given lesser priority. Examples of these include checking for statements which never can be executed and pinpointing storage errors due to faulty EQUIVALENCE usage. In the Appendix there is a full explanation of the rules not checked.

NON-PORTABLE FORTRAN CONSTRUCTS

Most FORTRAN compilers accept a superset of ANS FORTRAN. Mixed mode arithmetic and free format I/O are examples of common extensions of the Standard. These additional features are usually easy to spot.

Other features which may vary among compilers include constructs in FORTRAN where the method of implementation is not specified by the Standard, allowing various possibilities. For example, the Standard does not specify how argument passing is to be implemented (see Reference 3, sections 8.3.2, 8.4.2). There are situations in interprogram-unit communication where the results depend on the argument passing method. We shall refer to such function or subroutine calls as 'unsafe references'. They are illegal references according to the Standard, but are not detected by compilers, which usually view individual program units in isolation. A detailed description of the three types of unsafe references is given in the Appendix.

Here we shall present some examples of unsafe references.[8] We shall refer to two common methods of argument passing: passing by reference (i.e. passing an address) and by 'copy-in copy-out' (i.e. upon entry to a subprogram, copy an initial value for the argument into a temporary location to be used throughout the execution of the subprogram; upon return, copy the final contents of the temporary back into the actual argument).

Consider the following program fragment.

```
⋮
K = 2
CALL JOE (K, K)
⋮
END

SUBROUTINE JOE (M, N)
M = N * N * N
WRITE (7) M, N
⋮
END
```

Example 1

If both arguments are passed by reference, the values of M and N will both be 8 at the WRITE statement in Example 1. However, if arguments are passed by copy-in copy-out, the values of M and N will be 8 and 2 respectively. An analogous problem can arise if an array and an array element from that array appear in an actual argument list.

A more startling example of an unsafe reference appears in Example 2.

```
COMMON/JR/L
L = 0
CALL COUNT (L)
⋮
END

SUBROUTINE COUNT (M)
COMMON/JR/L
L = L+1
RETURN
END
```

Example 2

If the arguments are passed by reference the subroutine COUNT increments the common variable L each time it is called. However, in a copy-in copy-out situation, the mere presence of the dummy argument negates the incrementation even though the dummy argument is never used! That is, L is equal to 0 upon entry to COUNT; upon return to the main program from COUNT, L will be restored to 0 by the copy-out mechanism.

Generally, unsafe references which appear in actual FORTRAN programs are difficult to spot as they often involve several levels of subprogram references.

Another cause of errors in many FORTRAN programs involves the improper usage of common blocks to communicate between program units. The rule governing legal usage appears in the Appendix. Consider the following FORTRAN main program and two subroutines which illustrate the basic problem.

```
READ (10) X, Y
CALL TEST1 (X, Y)
CALL TEST2 (X, Y)
⋮
END

SUBROUTINE TEST1 (X, Y)
COMMON/OK/I
⋮
I = 0
IF (X .EQ. Y) I = 1
⋮
END

SUBROUTINE TEST2 (X, Y)
COMMON/OK/I
IF(I .EQ. 0) GOTO 4
⋮
END
```

Example 3

Subroutines TEST1 and TEST2 use the common block /OK/ to communicate with one another, however, /OK/ does not appear in the main program which calls them both. The Standard notes that if a function or subroutine contains a named common block that is *not* contained in any program unit currently referencing that program unit directly or indirectly, then, when control is returned to the calling program unit, the entities in the named common become undefined (see Reference 3, section 10.2.5). Any scheme for allocating named COMMON dynamically or for overlaying program units would invalidate the communication desired in Example 3. This usage is therefore illegal according to the Standard. In a more complex calling hierarchy, this usage becomes extremely difficult to check manually.

EXPERIENCE AND CONCLUSIONS

The PFORT Verifier has been used at Bell Laboratories to check several software packages including itself.[9, 10] Many of these programs have been installed already on a variety of computers; this experience attests to the portability of programs adhering to the rules of PFORT.

The Verifier has been installed successfully on the Honeywell 6000, the CDC 7600, the IBM 360 and the Univac 1108. The only FORTRAN changes (from the HIS 6000 version) made to install the Verifier on the CDC 7600 was to properly initialize a variable containing the number of characters per word and the variables used as I/O logical units.

Although speed was not a primary design goal, the Verifier processed itself at the approximate speed of 20 lines per second. This speed can be improved by recoding some of the lower level subprograms into assembly language for a particular installation.

PFORT and the PFORT Verifier are valuable tools for the writer of portable FORTRAN software. PFORT is a large portable subset of ANS FORTRAN whose definition helps to clarify the Standard. The Verifier renders PFORT more than a mere language definition by offering an automatic method of checking adherence. We conclude that PFORT and the Verifier are significant contributions to the field of language portability.

ACKNOWLEDGEMENTS

I wish to thank all those at Bell Laboratories who helped me in this project. I am most grateful to A. D. Hall, Jr. for his considerable assistance throughout all stages of this project. I am also grateful to M. D. McIlroy for his syntax of ANS FORTRAN, and to W. S. Brown, A. J. Goldstein and J. L. Ryder for their help with this paper.

APPENDIX

This Appendix defines PFORT, the portable subset of ANS FORTRAN. The definition presented here is in four parts. Firstly, we give a graphical syntax for PFORT which is a modified version of the full ANS FORTRAN syntax written by McIlroy.[6] The graphical syntax is followed by supplemental restrictions, numbered according to the syntax rules to which they apply. Secondly, we describe in detail the features of ANS FORTRAN which are not included in PFORT. Thirdly, we describe the interprogram-unit communications checked by the Verifier. Fourthly, we present the ANS FORTRAN rules not checked by the Verifier.

PFORT: annotated syntax

The annotated syntax presented in this section consists of graphical syntax charts with preliminary interpretive comments, notes supplementing the charts and a cross reference table for all entities in the charts.

The names of syntactic categories were chosen to suggest terms used in ANS FORTRAN. One exception is that 'element' is used in the syntax in places where the Standard refers to 'variable or array element'. Strings of lower case letters and properly imbedded hyphens denote syntactic categories. All other characters are literals.

There are five primitive syntactic categories (1–5) which are defined as follows:

letter any character from the set ABCDEFGHIJKLMNOPQRSTUVWXYZ

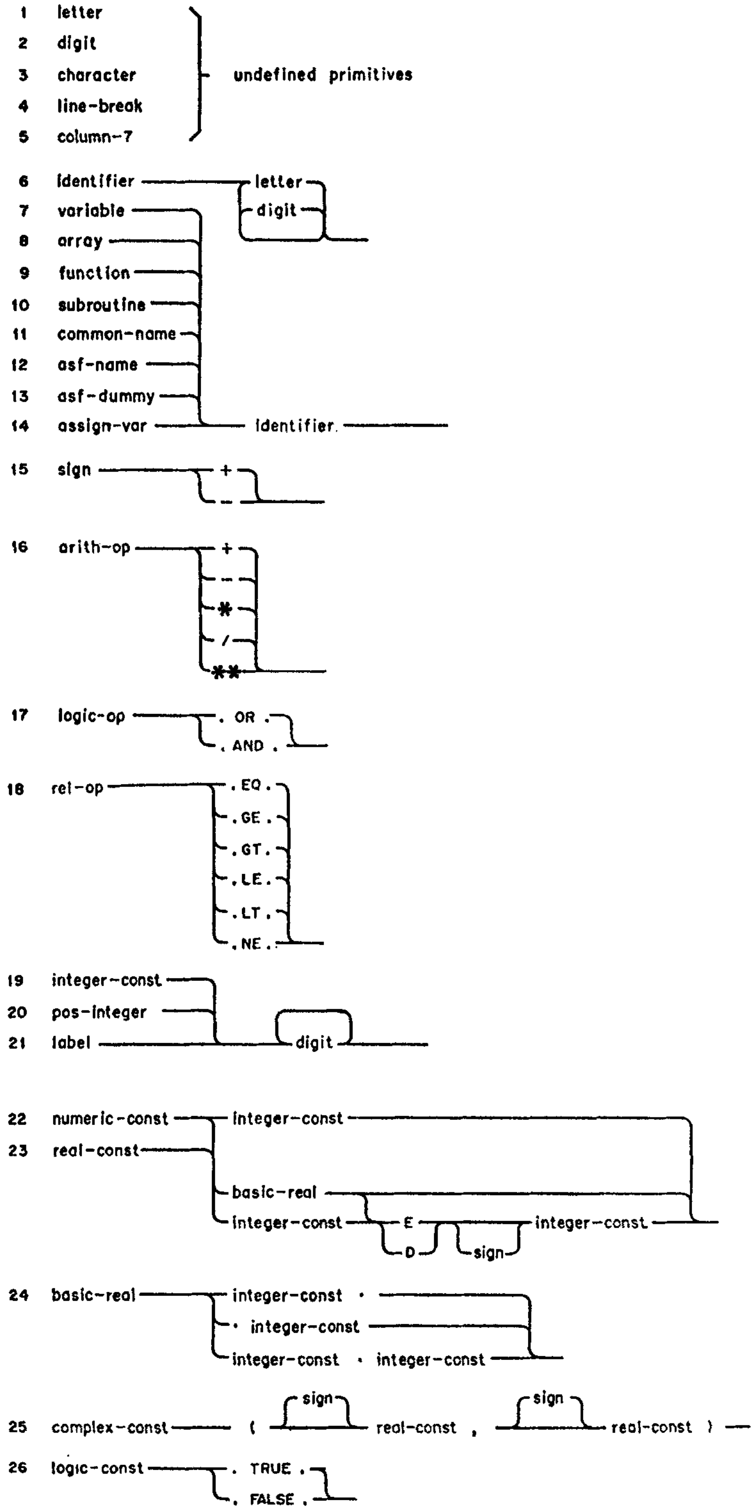
digit any character from the set 0123456789

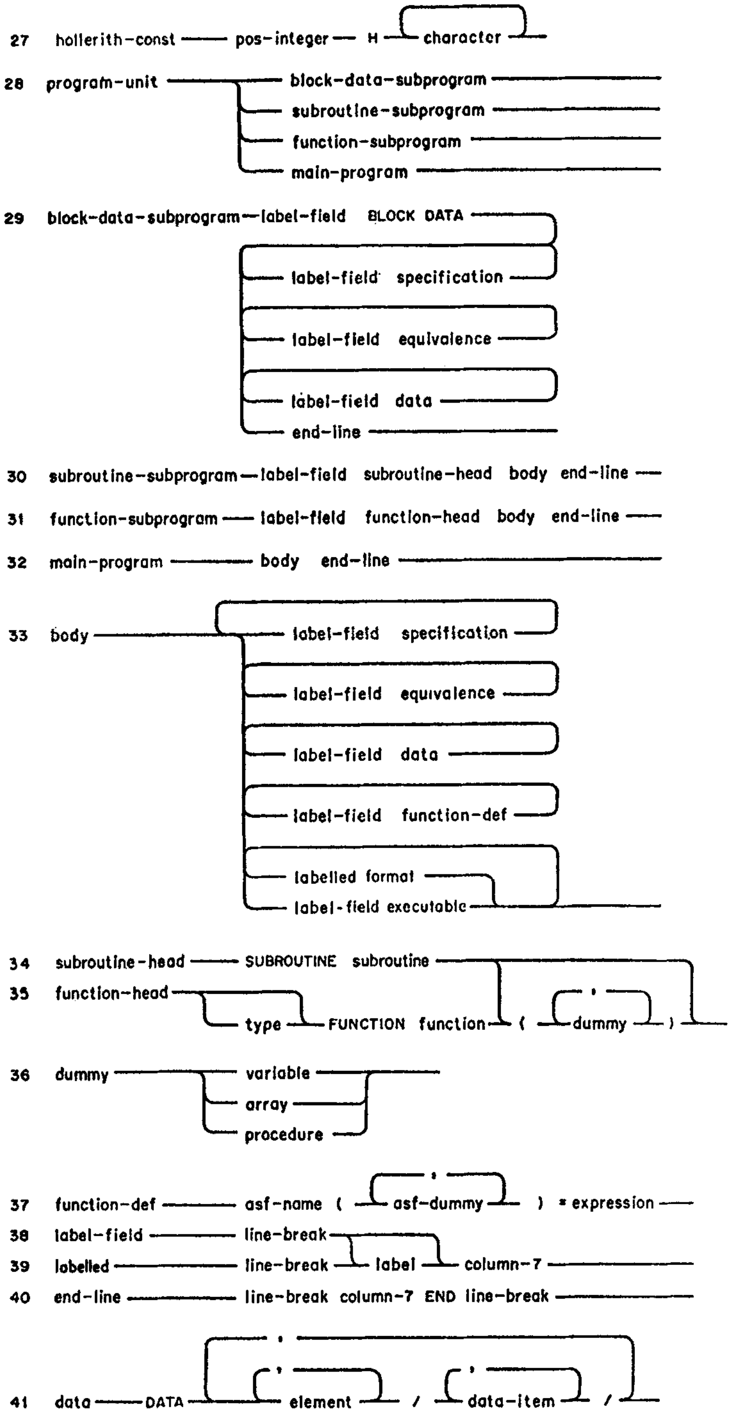
character any letter, digit, or character from the set − + = * / ( ) , .

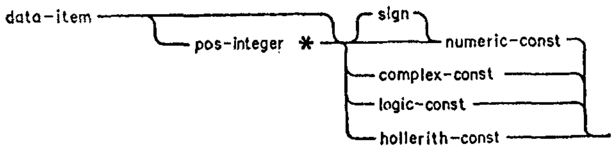
line-break the beginning or end of a FORTRAN initial line with all its continuation lines, or of an end line; comment lines are absorbed into line breaks

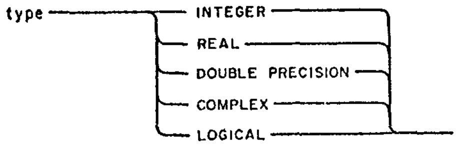
column-7 the beginning of a statement proper; column-7 absorbs the blank or 0 in column-6 of the initial line of that segment

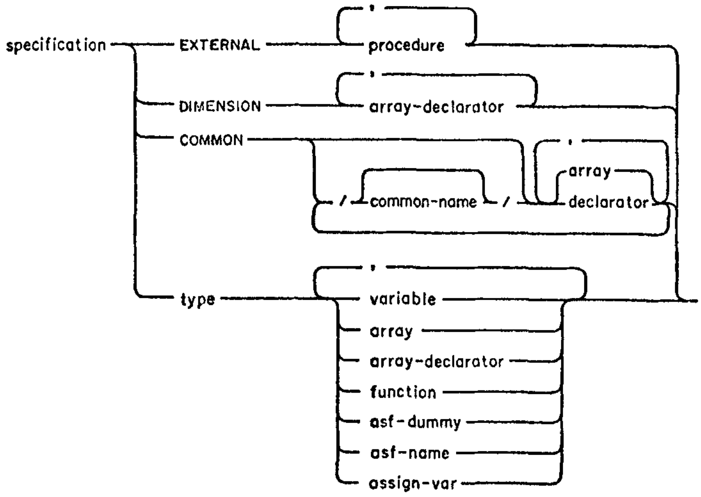
Blanks are not significant unless they appear within a Hollerith constant or a Hollerith field descriptor. Although continuation lines and comment lines are not considered in this syntax, the Verifier checks for their correct usage (see Reference 3, section 3.4).

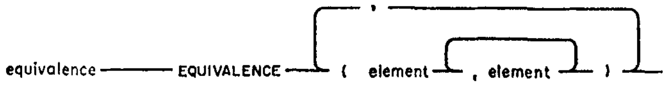

1 letter
2 digit
3 character
4 line-break
5 column-7
undefined primitives
6 identifier
letter
digit
7 variable
8 array
9 function
10 subroutine
11 common-name
12 asf-name
13 asf-dummy
14 assign-var
identifier
15 sign
+
-
16 arith-op
+
-
*
/
**
17 logic-op
. OR .
. AND .
18 rel-op
. EQ .
. GE .
. GT .
. LE .
. LT .
. NE .
19 integer-const
20 pos-integer
21 label
digit
22 numeric-const
23 real-const
integer-const
basic-real
integer-const
E
D
sign
integer-const
24 basic-real
integer-const .
. integer-const
integer-const . integer-const
25 complex-const
(
sign
real-const
,
sign
real-const
)
26 logic-const
. TRUE .
. FALSE .

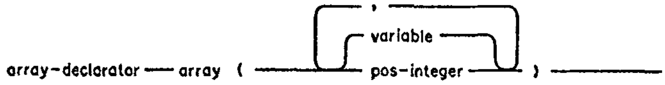

27 hollerith-const pos-integer H character
28 program-unit block-data-subprogram subroutine-subprogram function-subprogram main-program
29 block-data-subprogram label-field BLOCK DATA label-field specification label-field equivalence label-field data end-line
30 subroutine-subprogram label-field subroutine-head body end-line
31 function-subprogram label-field function-head body end-line
32 main-program body end-line
33 body label-field specification label-field equivalence label-field data label-field function-def labelled format label-field executable
34 subroutine-head SUBROUTINE subroutine
35 function-head type FUNCTION function (dummy ,)
36 dummy variable array procedure
37 function-def asf-name (asf-dummy ,) = expression
38 label-field line-break
39 labelled line-break label column-7
40 end-line line-break column-7 END line-break
41 data DATA element , / data-item , / ,

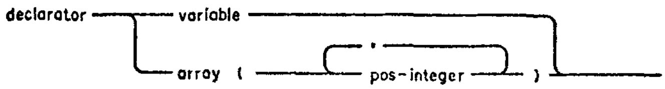
42 

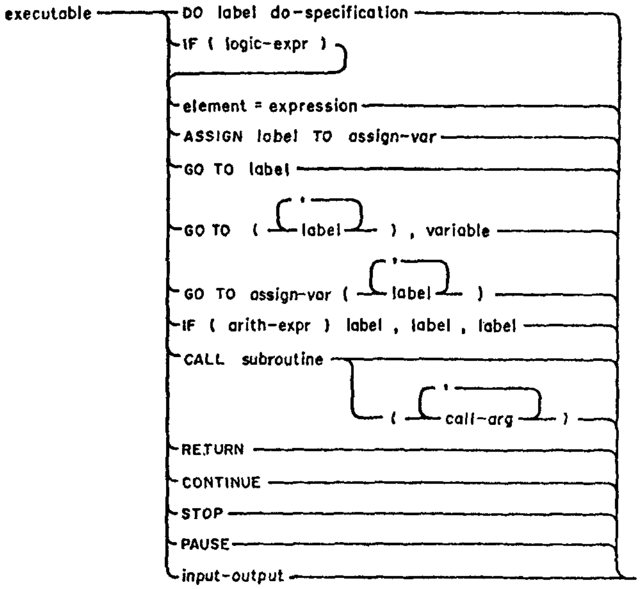
43 

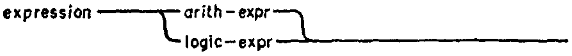
44 

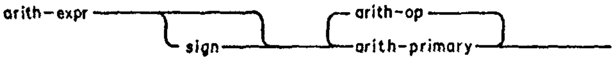
45 

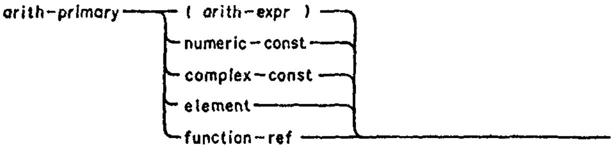
46 

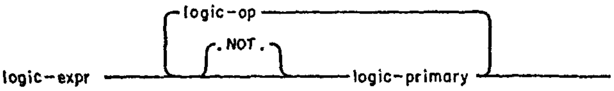
47 

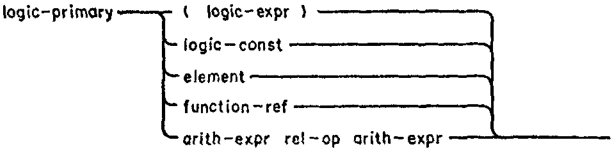
48 
executable
DO label do-specification
IF (logic-expr)
element = expression
ASSIGN label TO assign-var
GO TO label
GO TO (label) , variable
,
GO TO assign-var (label)
,
IF (arith-expr) label , label , label
CALL subroutine
(call-arg)
,
RETURN
CONTINUE
STOP
PAUSE
input-output

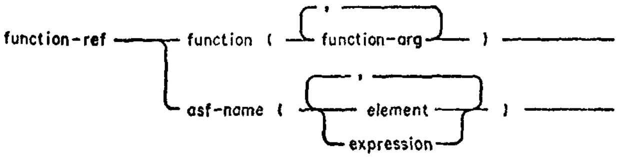
49 

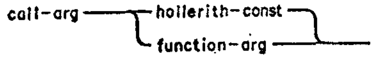
50 

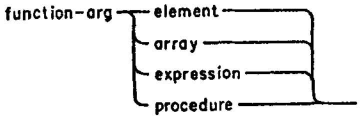
51 

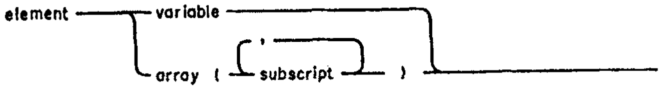
52 

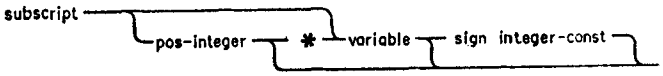
53 

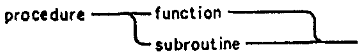
54 

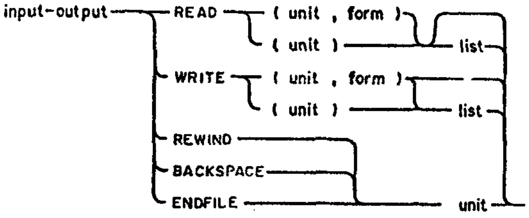
55


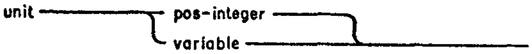
56


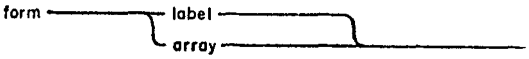
57


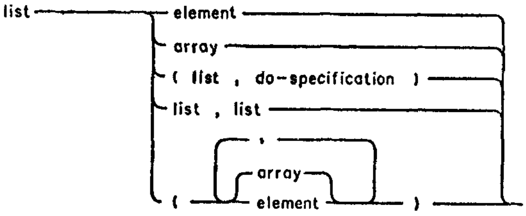
58


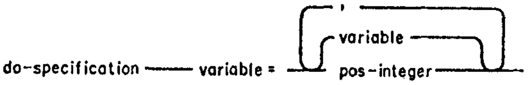
59


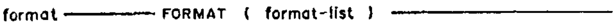
60


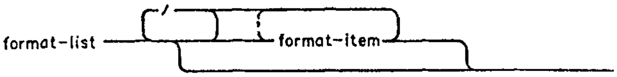
61


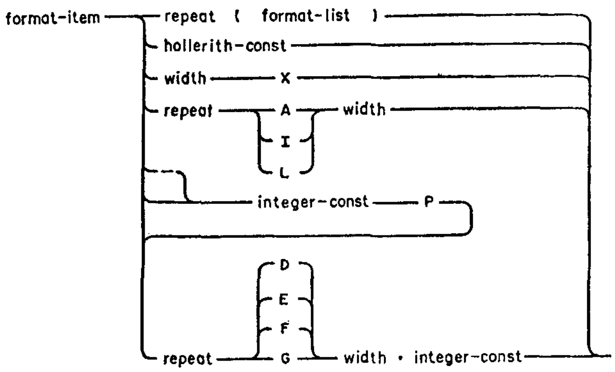
62


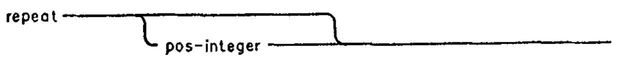
63


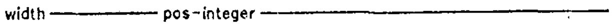
64


65


66


67


68


69


The following notes supplement the syntax. Each note is numbered with the number of the syntax rule to which it refers and is marked with an asterisk if it is more restrictive than ANS FORTRAN.

6. An identifier contains at most six letters and digits.

20. A pos-integer contains at least one non-zero digit.

21. A label contains at most five digits and at least one non-zero digit. Leading zeros are treated as blanks.

23. A real-const which contains the letter D is DOUBLE PRECISION.

25. Neither real-const can be DOUBLE PRECISION (see 23).

27. The pos-integer gives the number of characters in the hollerith-const.

29. EXTERNAL specifications cannot appear in a block-data-subprogram.

29. A common block appearing in a block-data-subprogram requires a common-name.

29.* One and only one block-data-subprogram can initialize a given common block. (This rule avoids the possibility that initialization of named common may depend upon the order of loading the subprograms.)

30. A subroutine-subprogram contains at least one RETURN statement.

31. A function-subprogram contains at least one RETURN statement.

32. A RETURN statement cannot appear in a main-program.

33. There must be at least one executable statement in a body.

35. Within a function-subprogram the function is used as if it were a variable. It must be assigned a value (see 41, 44, 45, 56, 64).

36. The procedure cannot be the function or subroutine which names this program-unit.

37. No array elements can appear in the expression. An asf-dummy of this arithmetic statement function can appear in the expression as if it were a variable. The type of the expression must be compatible with the type of the arithmetic statement function (see assignment in 48).

37.* In a function-def, a referenced external function may not change any of its arguments if one of the actual arguments is an asf-dummy.

38, 39. Two statements in a program-unit cannot have the same label.

40. An end-line is a single line with no continuation lines.

41. Neither a dummy nor the function which names this program-unit (see 35) can appear in a DATA statement.

41. A variable in named common or a variable equivalenced to a variable in named common may appear in a DATA statement if, and only if, the DATA statement is in a block-data-subprogram.

41.* A data-item must be the same type as the corresponding element except a Hollerith data-item must correspond to an INTEGER element.

41. The number of data-items, counting replication, must equal the number of elements.

41.* If the element is an array element it contains only pos-integer subscripts.

42.* A hollerith-const contains only one character. (This preserves independence from word length of the machine.)

44. A given procedure can appear in an EXTERNAL statement only once within a program-unit.

44. Neither the procedure in an EXTERNAL statement nor the function (see 35) in a type statement can be the function or subroutine which names this program-unit.

44. Neither a dummy nor the function which names this program-unit (see 35) can appear in a COMMON statement.

44. A variable or an array cannot appear in more than one common block within a program-unit.

44. If a common-name appears more than once in a program-unit or within a COMMON statement, then the associated common entries are strung together in the order of their appearance.

44.* Within a common block, COMPLEX and DOUBLE PRECISION entries must precede all REAL, LOGICAL or INTEGER entries. (This prevents improper word alignment of data.)

44. No variable, array, function, asf-name, asf-dummy or assign-var can be explicitly typed more than once in a program-unit. Such entities have implicit types, unless they appear in a type statement. If their first letter is I, J, K, L, M, N they are type INTEGER; otherwise they are REAL.

45. Neither a dummy nor the function which names this program-unit (see 35) can appear in an EQUIVALENCE statement.

45. Elements contained in two different common blocks cannot be equivalenced to each other (either directly or indirectly).

45. If the element is an array element it must have pos-integer subscripts.

45.* Corner elements [e.g. a(1, 1, 1)] must be used to equivalence elements of different data types.

45.* The previously declared number of subscripts must be used in an array element.

46. An array-declarator contains at most two commas.

46. If an array-declarator contains a variable, the array must be a dummy, the variable must be a dummy of type INTEGER and the value of the variable must not be changed by any statement in the program-unit.

46, 47. An array can appear only once in an array-declarator or declarator within a program-unit (see 44).

47. A declarator contains at most two commas.

48.* The label in a DO statement must appear after this DO statement as the label of an executable statement other than a STOP, PAUSE, RETURN, GOTO, IF or DO statement.

48.* The range of a DO statement is all executable statements from and including the first executable statement following the DO, to and including the terminal statement associated with the DO (i.e. the statement whose label matches the label in the DO statement). If the range of a DO statement contains another DO statement, the range of the contained DO must be a subset of the range of the containing DO. The values of variables used in the do-specification cannot be changed within the range of the DO. (Note: 'extended range' is not permitted in PFORT.)

48. In assignment, the type of the expression must be compatible with the type of the element. That is, if the element is of type INTEGER, REAL or DOUBLE PRECISION, then the expression must be one of these three types; if the element is COMPLEX, then the expression must be COMPLEX; if the element is LOGICAL, then the expression must be LOGICAL.

48. In GOTO and ASSIGN statements, the assign-var or the variable must be of type INTEGER.

48. In arithmetic IF statements, the arith-expr cannot be COMPLEX.

48.* The PAUSE statement, although within PFORT, is flagged by the Verifier. (It has different meaning on different operating systems.)

48.* In all forms of GOTO and IF statements, the labels must be such that there can be no transfer into the range of a DO.

50. An arith-primary must not be both preceded and followed by the operator **.

50. In evaluating an arith-expr, the order of evaluation of the arith-primaries is optional. The mathematical properties of each arith-op can be used to re-order the evaluation of the expression as long as parentheses are not violated. The precedence of the arithmetic operators from highest to lowest is: ** {* /} {+ −}.

50. An arith-primary can be legally combined with another arith-primary using an arith-op (except **) according to the following rules:

(a) INTEGER op INTEGER, result is INTEGER.

(b) REAL op REAL, result is REAL.

(c) COMPLEX op COMPLEX, result is COMPLEX.

(d) DOUBLE PRECISION op DOUBLE PRECISION, result is DOUBLE PRECISION.

(e) REAL op DOUBLE PRECISION or DOUBLE PRECISION op REAL, result is DOUBLE PRECISION.

(f) REAL op COMPLEX or COMPLEX op REAL, result is COMPLEX.

50. An arith-primary can be legally combined with another arith-primary using '**' according to the following rules:

(a) INTEGER ** INTEGER, result is INTEGER.

(b) REAL ** INTEGER, REAL ** REAL, result is REAL.

(c) DOUBLE PRECISION ** INTEGER, DOUBLE PRECISION ** REAL, DOUBLE PRECISION ** DOUBLE PRECISION, REAL ** DOUBLE PRECISION, result is DOUBLE PRECISION.

(d) COMPLEX ** INTEGER, result is COMPLEX.

52. The logical operator .AND. has higher precedence than the logical operator .OR..

53. An arith-expr can be combined with another arith-expr using a rel-op to form a logic-primary according to the following rules:
(a) Both INTEGER.
(b) Both REAL.
(c) Both DOUBLE PRECISION.
(d) One REAL and the other DOUBLE PRECISION.

56. The procedure cannot be the function or subroutine which names this program-unit.

57. An element contains at most two commas.

58. The variable must be of type INTEGER.

60.* The BACKSPACE statement, although within PFORT, is flagged by the Verifier. (It has a different meaning on different operating systems.)

61. The variable must be of type INTEGER.

62. The label must appear as the label of a FORMAT statement in the program-unit.

62.* The array must be of type INTEGER.

63. The values of the variables used in the do-specification cannot be changed within the parentheses associated with that do-specification.

64. A do-specification contains one or two commas.

64. A variable must be of type INTEGER.

64.* In a function-subprogram the function cannot be used as the control variable (to the left of '=') in a do-specification (see 35).

65.* A FORMAT statement contains one or two levels of parentheses.

67.* The width in A format must be 1. (This preserves independence from the word length of the machine.)

Table I. Cross references

Entry	Defined	Used in
A		67
arith-expr	50	48, 49, 51, 53
arith-op	16	50
arith-primary	50	51
array	8	36, 44, 46, 47, 56, 57, 62, 63
array-declarator	46	44
asf-dummy	13	37, 44
asf-name	12	37, 44, 54
ASSIGN...TO		48
assign-var	14	44, 48
BACKSPACE		60
basic-real	24	22, 23
BLOCK DATA		29
block-data-subprogram	29	28
body	33	30, 31, 32
CALL		48
call-arg	55	48
character	3	27
column-7	5	38, 39, 40
COMMON		44

Table I—*continued*

Entry	Defined	Used in
common-name	11	44
COMPLEX		43
complex-const	25	42, 51
CONTINUE		48
D		22, 23, 67
DATA		41
data	41	29, 33
data-item	42	41
declarator	47	44
digit	2	6, 19, 20, 21
DIMENSION		44
DO		48
do-specification	64	48, 63
DOUBLE PRECISION		43
dummy	36	34, 35
E		22, 23, 67
element	57	41, 45, 48, 51, 53, 54, 56, 63
END		40
ENDFILE		60
end-line	40	29, 30, 31, 32
EQUIVALENCE		45
equivalence	45	29, 33
executable	48	33
expression	49	37, 48, 54, 56
EXTERNAL		44
F		67
form	62	60
FORMAT		65
format	65	33
format-item	67	66
format-list	66	65, 67
FUNCTION		35
function	9	35, 44, 54, 59
function-arg	56	54, 55
function-def	37	33
function-head	35	31
function-ref	54	51, 53
function-subprogram	31	28
G		67
GOTO		48
H		27
hollerith-const	27	42, 55, 67
I		67
identifier	6	7, 8, 9, 10, 11, 12, 13, 14
IF		48
input-output	60	48
INTEGER		43
integer-const	19	22, 23, 24, 58, 67
L		67
label	21	38, 39, 48, 62
label-field	38	29, 30, 31, 33
labelled	39	33
letter	1	6

Table I—*continued*

Entry	Defined	Used in
line-break	4	38, 39, 40
list	63	60, 63
logic-const	26	42, 53
logic-expr	52	48, 49, 53
logic-op	17	52
logic-primary	53	52
LOGICAL		43
main-program	32	28
numeric-const	22	42, 51
P		67
PAUSE		48
pos-integer	20	27, 42, 46, 47, 58, 61, 64, 68, 69
procedure	59	36, 44, 56
program-unit	28	
READ		60
REAL		43
real-const	23	25
rel-op	18	53
repeat	68	67
RETURN		48
REWIND		60
sign	15	22, 23, 25, 42, 50, 58
specification	44	29, 33
STOP		48
SUBROUTINE		34
subroutine	10	34, 48, 59
subroutine-head	34	30
subroutine-subprogram	30	28
subscript	58	57
type	43	35, 44
unit	61	60
variable	7	36, 44, 46, 47, 48, 57, 58, 61, 64
width	69	67
WRITE		60
X		67
.		24, 67
.AND.		17
.EQ.		18
.FALSE.		26
.GE.		18
.GT.		18
.LE.		18
.LT.		18
.NE.		18
.NOT.		52
.OR.		17
.TRUE.		26
=		37, 48, 64
+		15, 16
−		15, 16, 67
*		16, 42, 58
**		16
/		16, 41, 44, 66

Table I—*continued*

Entry	Defined	Used in
,		25, 34, 35, 37, 41, 44, 45, 46, 47, 48, 54, 57, 60, 63, 64, 66
(...)		25, 34, 35, 37, 45, 46, 47, 48, 51, 53, 54, 57, 60, 63, 65, 67

PFORT: differences from ANS FORTRAN

In the preceding charts, notes with asterisks delineate rules of PFORT which are restrictions of ANS FORTRAN. There are some restrictions which cannot be easily shown in that manner. They are listed here with references to the charts where applicable.

(1) '$' is not in the character set (see rule 3).

(2) Common-name, asf-name, assign-variable, dummy and asf-dummy are required to be distinct unique usages of identifiers within a program unit (see rules 11–14, 59).

(3) The octal-constant is not in PFORT.

(4) The sequential ordering of statements in a program unit is restricted in PFORT (see rule 33). The more general ordering allowed in ANS FORTRAN is as follows:

(a) Specification statements, EQUIVALENCE statements, FORMAT statements, or none of these.

(b) DATA statements, FORMAT statements, arithmetic statement function definitions, or none of these.

(c) DATA statements, FORMAT statements, or none of these, and at least one executable statement.

(d) END statement.

(5) The extended range DO statement is not in PFORT (see rule 48).

PFORT: interprogram-unit interfaces

Here is a description of the interprogram-unit communication rules which are checked by the Verifier:

(1) Common.

(a) The total length of a named common block must match in all occurrences of that common block in the program. The lengths of the DOUBLE PRECISION and COMPLEX entries must agree. (Note: EQUIVALENCE can be used to circumvent this restriction.)

(b) Among all program units containing a particular common block /B/, there must exist a program unit A, such that one must execute a nested sequence of subprogram references beginning in A, in order to reach any other program unit containing /B/.

(2) Subprogram references.

(a) The number of actual arguments in a reference (see rules 55, 56) must agree with the number of dummy arguments in the subprogram definition.

(b) The types of the actual arguments in a reference must match the types of the corresponding dummy arguments in the subprogram definition. Hollerith actual arguments must correspond to dummy INTEGER arrays.

(c) The structures of the actual arguments in a reference must match those of the corresponding dummy arguments. In particular, a scalar actual argument must match a scalar dummy argument and an array actual must match an array dummy. An array element actual may match either a scalar dummy or an array dummy. Some non-standard compilers do not allow an array element actual to become associated with an array dummy.

(d) Unsafe references.

We assume that array arguments are always passed by reference. We assume a value of an identifier 'can be changed' if it appears within the program unit to the left of an equals sign in an assignment statement or as an argument whose value can be changed in a subprogram reference.

(i) If an actual argument is an expression or constant and the corresponding dummy argument can be changed, then the reference is unsafe.

(ii) If an actual argument appears twice in an argument list, one of the corresponding dummy arguments is a scalar, and one of the corresponding dummy arguments can be changed, then the reference is unsafe. (An array and an array element of that array or two array elements of the same array are considered to be the same actual argument appearing twice)

(iii) If an actual argument in common is associated with a scalar dummy, the called subprogram has access to the same COMMON block (directly or indirectly), and the called subprogram changes either the dummy or the COMMON block then the reference is unsafe. (Note this is the probable intent of Reference 3, sections 8.3.2, 8.4.2.)

(3) Basic external functions and intrinsic functions.

The Verifier recognizes basic external and intrinsic functions (see Reference 3, sections 8.2, 8.3.3). It checks references to them for adherence to default argument types and number (see rule 2). If a basic intrinsic is redefined by a program, its usage is checked for consistency with the new definition. Correct usage of basic external and intrinsic functions is governed by Reference 3, sections 8.3.2, 10.1, 10.1.7, 10.1.8. If usage is not consistent throughout the program, an error diagnostic will be produced.

PFORT: ANS FORTRAN rules not checked by the Verifier

(1) Errors which occur through misuse of arithmetic operators.

For example, a negative valued arith-primary cannot be raised to a REAL or DOUBLE PRECISION exponent (see Reference 3, section 6.4).

(2) Rules which can only be checked, if at all, by extensive analysis of the program or at execution time.

The following are examples of these:

(a) When a computed GOTO statement is executed, the variable must be positive and less than or equal to the number of labels in the parenthesized list (see Reference 3, section 7.1.2.1.3).

(b) When a DO statement is executed, the variables in the do-specification must be positive (see Reference 3, section 7.1.2.8).

(c) The evaluation of a function reference cannot alter the value of any other primary in the statement in which the reference appears (see Reference 3, section 6.4).

(d) No array element may contain a subscript that, during execution, assumes a value less than one or greater than the maximum length implied by the array declarator (see Reference 3, section 7.2.1.1).

(3) Rules concerning the defining and undefining of variables.

For example, the Standard prohibits the use of undefined variables in expressions. A variable can become undefined in many ways. The control variable of a DO loop is defined on branching out of the DO by a GOTO statement, however it is undefined after exiting by completion of the DO (see Reference 3, sections 7.1.2.8, 10.2).

(4) Rules concerning second level definition.

For example, INTEGER variables used as subscripts or in computed GOTO statements must be defined at the second level (see Reference 3, section 10.2.8).

(5) Rules involving storage allocation.

The following are examples of such rules:

(a) It is illegal for EQUIVALENCE to result in two different elements of an array being stored in the same location (see Reference 3, section 7.2.1.4).

(b) A storage unit may only be initialized once in a executable program (see Reference 3, section 10.1.3).

(c) The use of variables of different types which share the same storage location through use of EQUIVALENCE is restricted by the Standard (see Reference 3, section 10.2).

(6) Rules concerning the compatibility of I/O list elements with their corresponding format fields and those concerning I/O conversions (see Reference 3, sections 7.2.3.4, 7.2.3).

REFERENCES

1. 'Clarifications of FORTRAN standards—initial progress', *Comm. ACM*, **12**, 289–294 (1969).
2. 'Clarification of FORTRAN standards—second report', *Comm. ACM*, **14**, 628–642 (1971).
3. *American National Standard FORTRAN*, American National Standards Institute, New York, 1966.
4. A. D. Hall, Jr. and B. G. Ryder, *Installation of the FORTRAN Verifier*, Bell Telephone Laboratories, Murray Hill, New Jersey, 1973.
5. A. D. Hall, Jr., *A Portable FORTRAN IV Subset*, Bell Telephone Laboratories, Murray Hill, New Jersey, 1969, unpublished.
6. M. D. McIlroy, *ANS FORTRAN Charts*, Computing Science Technical Report No. 13, Bell Telephone Laboratories, Murray Hill, New Jersey, 1971, unpublished.
7. W. S. Brown, *ALTRAN User's Manual*, Bell Telephone Laboratories, Murray Hill, New Jersey, 1971.
8. B. G. Ryder, *The FORTRAN Verifier: Motivation and Implementation*, Bell Telephone Laboratories, Murray Hill, New Jersey, 1972, unpublished.
9. A. D. Hall, Jr., *The M6 Macro Processor*, Computing Science Technical Report No. 2, Bell Telephone Laboratories, Murray Hill, New Jersey.
10. B. G. Ryder, *The FORTRAN Verifier: User's Guide*, Computing Science Technical Report No. 12, Bell Telephone Laboratories, Murray Hill, New Jersey.