Computing Science Technical Report #12

The PFORT Verifier

B. G. Ryder
A. D. Hall

Part 1

May 1973
Revised: July 1975
August 1978
April 1979
January 1981

B. G. Ryder
The PFORT Verifier: User's Guide

Part 2

A. D. Hall and B. G. Ryder
Installation of the PFORT Verifier

Part 1 is a revised version of B. G. Ryder's paper, "The PFORT Verifier," *Software Practice and Experience 4*, pages 359-377 (October 1974), published by John Wiley & Sons, Ltd., Baffins Lane, Chichester, Sussex, England.

The PFORT Verifier: User's Guide

B. G. Ryder
Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

The PFORT Verifier is a program which checks a FORTRAN program for adherence to PFORT, a portable subset of American National Standard FORTRAN. It diagnoses errors in inter-program-unit communication and COMMON usage which compilers often miss. The Verifier itself is written in PFORT and can easily be installed on a variety of computers. This paper describes the use of the Verifier and presents the portable subset in considerable detail.

The PFORT Verifier: User's Guide

B. G. Ryder
Bell Laboratories
Murray Hill, New Jersey 07974

I. INTRODUCTION

The PFORT Verifier is a program which checks a FORTRAN program for adherence to PFORT, a portable subset of American National Standard FORTRAN [1,2,3,4]. Most FORTRAN compilers check only that individual program units conform to the Standard; the Verifier also checks that inter-program-unit communication through the use of COMMON and argument lists is consistent with the Standard.

The Verifier provides a number of facilities useful in debugging and documentation. It produces intra-program-unit error diagnostics, symbol tables, and cross references. The inter-program-unit output includes for each program unit, a listing of the arguments, the common blocks, the program units called, and the program units which call it. A list of global common definitions is also produced.

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. This is described in detail in Part 2 of Reference [6].

Section II of this paper briefly describes the classes of errors which the Verifier diagnoses. Sections III, IV, and V describe the use of the Verifier. A complete definition of PFORT appears in Section VI.

This paper is designed to facilitate use of the Verifier. Reference [5] can be consulted for a discussion of the development of PFORT, for further examples of the non-portable FORTRAN constructs discussed in Section II, and for a summary of our experience with PFORT and the PFORT Verifier.

II. ERRORS DIAGNOSED BY THE VERIFIER

There are two classes of errors diagnosed by the Verifier. The first class includes the syntax errors usually found by a compiler (e.g., illegal punctuation in a FORTRAN statement, missing END statement, etc.). The second class is comprised of errors in inter-program-unit communications. These are not detected at all by most compilers; yet they cause much difficulty in the writing of truly portable software. Included in this class of errors are improper matching of actual arguments with dummy arguments in subprogram references, recursion, illegal common usage, and *unsafe* references. Illegal common usage and unsafe references are discussed briefly below and in detail in Section VI. Problems caused by these errors are further illustrated in Reference [5].

The rules for legal usage of common insure that all definitions of named common blocks are consistent and that common blocks are used properly in inter-program-unit communication. These rules prevent difficulties with word alignment and dynamic vs. static methods of storage allocation.

Unsafe references involve situations in argument passing which may yield implementation dependent results. The Standard is written in such a way as to make *illegal* any program which could reveal the difference in the actual implementation of argument passing [1, Sections 8.3.2,

8.4.2]. Here, we will show how to pick the illegal references out of the set of unsafe references flagged by the Verifier.

There are three types of unsafe references found by the Verifier. They are unsafe because different implementations of argument passing may lead to different results. One must examine the flow of control of the program with the given data in order to determine which unsafe references are truly illegal.

In "Inter-program-unit Interfaces" (in Section VI) we carefully define the three types of unsafe references which are illustrated below. As explained there, the Verifier does not perform a flow-of-control analysis of the program. Therefore, in identifying unsafe references, the Verifier assumes that a value which can be changed *is* changed by the program. Unsafe references are only illegal if the values which "can be changed" are *in fact* changed during the execution of the program. Nevertheless, unsafe references are, in general, programming constructs which should be avoided for maximum safety.

In order to examine examples of unsafe references we shall consider the two commonly implemented methods of argument passing: passing by "reference" (i.e., passing by address) and passing 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). We assume that array arguments are always passed by reference.

The simplest and first type of unsafe reference occurs when a constant or expression is associated with a dummy argument whose value may be changed. Example 1 illustrates this.

```
1    CALL JON (-1)
     .
     .
     .
     END

     SUBROUTINE JON(J)
2    J=IABS(J)
     .
     .
     .
     END
```

Example 1

The reference at statement number 1 is unsafe because it may change the value of the constant -1, used as an actual argument. This reference is illegal as well, as a program cannot be allowed to reset the value of constants.

The second type of unsafe reference occurs when the same actual argument appears twice in an argument list, one of the corresponding dummies is a scalar, and either of the dummy arguments may be changed. Example 2.1 illustrates this type of unsafe reference.

```
1    CALL JON (X, X)
     .
     .
     .
     END

     SUBROUTINE JON(A, B)
2    A=ABS(A)
     .
     .
     .
     END
```

Example 2.1

The call of JON at statement number 1 will be noted as an unsafe reference by the Verifier. It actually is an illegal reference for negative values of X. It is illegal because subsequent to statement 2 in JON, if X has the value -1 in the call, then A and B will have the values 1, -1 if arguments are passed by copy-in, copy-out and 1, 1 if arguments are passed by reference. However, for non-negative values of X, the program is legal although poorly written.

Analyzing this type of unsafe reference can be much trickier if arrays are involved. If an array and an array element from that array *or* two array elements from the same array appear in an actual argument list, with one of those actuals corresponding to a scalar dummy argument whose value is changed, the reference will be flagged by the Verifier as a type 2 unsafe reference. Determining the legality (or illegality) of such a reference involves knowledge of the subscript values at the time of the reference as well as flow of control information. Since the Verifier does not have either, the user must carefully examine these unsafe references to determine which are illegal. Example 2.2 illustrates these references and shows how the legality of a reference can hinge on subscript values as well as flow-of-control information.

```
      DIMENSION A(10)
      .
      .
      .
1     CALL SAM (A(1), A, 10)
2     CALL SAM (A(1), A(4), 7)
3     CALL SAM (A(I), A(L), 11-L)
      .
      .
      .
      END

      SUBROUTINE SAM (X, Y, N)
      DIMENSION Y(N)

C     SAM SELECTS LARGEST ELEMENT
C     FROM SET (X, Y(1), Y(2), ..., Y(N))
C     SENDS VALUE BACK IN X

      DO 1 I=1,N
      IF (X.LT.Y(I)) X=Y(I)
1     CONTINUE
      END
```

Example 2.2

In Example 2.2, the reference at statement 1 causes both X and Y(1) to be associated with A(1). The Verifier will flag this reference as a type 2 unsafe reference. If A(1) is *not* the largest element of A, the assignment of X = Y(I) will occur in subroutine SAM for at least one value of I. Let us assume that A(5) = 10.0 is the largest element and that A(1) = -1.0. After the DO loop, X and Y(1) will have the values 10., 10. if the arguments are passed by reference; but will have the values 10., -1. if arguments are passed by copy-in, copy-out. Clearly this reference is illegal. If A(1) is the largest element in A, the reference is legal, because the assignment X = Y(I) will never occur. However, we can argue that one would not use SAM if one already knew A(1) was the largest element.

The reference at statement 2, flagged as unsafe by the Verifier, cannot result in one element of A being associated with both X and an element of Y; therefore, it is not possible to assign two values to such an element through its two associations with dummy arguments. Statement 2 is a legal reference. The Verifier calls this reference unsafe because it does not evaluate subscript values.

The reference at statement 3, flagged as unsafe by the Verifier, can be legal or illegal depending on the values of I and L. If I is less than L, we are in a situation similar to statement 2 where the same element of A cannot be associated with both X and an element of Y. The reference is legal. If I is greater than or equal to L, then the same element of A *can* be associated with both X and an element of Y. We must analyze the situation as we did with statement 1 to determine the legality of the reference. If A(I) is the largest element in {A(L), A(L+1), ..., A(10)}, the reference is legal; otherwise it is not.

The third type of unsafe reference occurs when an actual argument is in common and associated with a scalar dummy, and the called subprogram has access directly or indirectly to the common block and may change the corresponding dummy argument and/or common block. We illustrate this type of unsafe reference with Examples 3.1 and 3.2.

```
      COMMON /WSB/ L
      L = -2
1     CALL SUE(L)
      .
      .
      .
      END

      SUBROUTINE SUE(I)
10    I = IABS(I)
11    CALL SALLY(I)
      RETURN
      END

      SUBROUTINE SALLY(J)
      COMMON /WSB/ L
      .
      .
      .
      END
```

Example 3.1

In Example 3.1, the actual argument L is in common block /WSB/, SUE has access to /WSB/ indirectly and SUE sometimes changes the value of its dummy argument. The reference to SUE will be flagged as unsafe by the Verifier. The reference causes I in subroutine SUE to be associated with L in common block /WSB/. The value of I is changed from -2 to 2 by statement number 10 in SUE. The subsequent reference to subroutine SALLY results in J being associated with L in common block /WSB/. But upon entering SALLY, J = 2 and L = 2 if arguments are passed by reference and J = 2 and L = -2 if arguments are passed by copy-in, copy-out! Clearly, this is illegal. If SUE had had *no* indirect or direct access to common block /WSB/, the reference at statement number 1 would have been legal. Also, if SUE had been called with a nonnegative argument the reference would have been legal.

In Example 3.2, the actual argument L is in common block /WSB/, MARY contains /WSB/, and MARY changes the value L. The reference to MARY is flagged by the Verifier as unsafe.

```
COMMON /WSB/ L
L = 0
CALL MARY (L)
.
.
.
STOP
END

SUBROUTINE MARY (J)
COMMON /WSB/ L
L = L + 1
RETURN
END
```

Example 3.2

If arguments are passed by reference in Example 3.2, MARY increments the common variable L each time it is referenced. If the arguments are passed by copy-in, copy-out the presence of the dummy negates the incrementation! That is, L is 0 upon entry to MARY. L is incremented to 1 in MARY. The value of L will be restored to 0 upon return to the main program by the copy-out mechanism, even though the dummy argument is not used in MARY!

We should note that these unsafe references can be complicated by involving longer chains of subprogram references. One should attempt to avoid using unsafe references and thus guarantee the legality of inter-program-unit communications.

III. INPUT

The Verifier accepts as input a sequence of one or more program units comprising an executable FORTRAN program [1, Section 9.1.6]. Commands to the Verifier may be included as discussed in Section V. End-of-input is signaled by a record (card) containing a "." in column 1, although this record may be optional at some installations.

IV. OUTPUT

The Verifier produces a statement-by-statement listing of each program unit followed by an alphabetized symbol table which lists attributes and cross references for all the symbols in that program unit (see Figures 1, 2). If an error is discovered, an error message, preceded by three asterisks, is printed directly beneath the statement at which the error occurred (see Figure 1). Any errors discovered after the program unit has been listed are printed with the statement number and program unit where they occurred (see Figure 3). The global structure of the FORTRAN program is presented in an alphabetized list of all its program units. Notice the Verifier includes in this list basic external functions which appear in the program (e.g., SQRT in Figure 3). For each program unit, the Verifier lists its arguments, its common blocks, the program units it calls, and the program units which call it (see Figure 3). A table of global common block definitions is also produced (see Figure 3).

Key to Figures 1, 2

Type Key

column 1:

E explicitly typed

column 2:

I INTEGER
R REAL
D DOUBLE PRECISION
C COMPLEX
L LOGICAL
H HOLLERITH

Usage Key

column 1, 2:

FA arithmetic-statement-function argument
FN function name
E external (function or subroutine)
GT assigned goto variable
IF intrinsic function
SF arithmetic statement function
SN subroutine name
V variable

Attribute Key

column 1:

C in COMMON

column 2:

E in an EQUIVALENCE Statement

column 3:

A dummy argument

column 4:

S value set by program unit

column 5, 6:

S scalar
An array with n dimensions

```
PFORT VERIFIER 12/19/80 VERSION

 1                FUNCTION ENORM(N, X)
                  C
                  C   FUNCTION CALCULATES EUCLIDEAN NORM OF VECTOR X
                  C
 2                DIMENSION X(N)
 3                LOGICAL ERR
 4                COMMON /ERROR/ERR
 5                ENORM=0.0
 6                IF(N.LE.0) GOTO 3
 7                DO 1 R=1,N

*** CONTROL VARIABLE NOT INTEGER SCALAR

 8              1 ENORM=ENORM + X(R) * X(R)

*** ILLEGAL VARIABLE IN SUBSCRIPT CONSTRUCTION


*** ILLEGAL VARIABLE IN SUBSCRIPT CONSTRUCTION

 9                ENORM=SQRT(ENORM)
10              2 RETURN
11              3 ERR=.TRUE.
12                GOTO 2
13                END


PROGRAM UNIT    ENORM


ARGUMENTS       N     X

NAME     TYPE   USE   ATTRIBUTES   REFERENCES

ENORM       P   FN      S          1    5    8    9
ERR        EL   V     C SS         3    4   11
N           I   V      A S         1    2    6
R           R   V       S         7    8
SQRT       ER   FN                 9
X           R   V      A A1       1    2    8
1                                  7    8
2                                 10   12
3                                  6   11


COMMON BLOCKS

ERROR     ERR
```

Figure 1

```
PFORT   VERIFIER   12/19/80   VERSION

            C
            C       DRIVER PROGRAM TO TEST EUCLIDEAN NORM FUNCTION
            C
      1             INTEGER X(100)
      2             LOGICAL ERR
      3             COMMON /ERROR/ERR
      4       1     READ(5,10) I
      5      10     FORMAT (12I5)
            C       END OF DATA CHECK
      6             IF(I.GT.100) STOP
      7             READ(5,10) (X(J), J=1,I)
      8             ERR=.FALSE.
      9             ANS=ENORM(I, X)
     10             IF (.NOT.ERR) GOTO 2
     11             WRITE(6,20)
     12      20     FORMAT(15H BAD VALUE OF N)
     13             GOTO 1
     14       2     WRITE(6,30) ANS
     15      30     FORMAT(6H NORM=, E15.7)
     16             GOTO 1
     17             END


PROGRAM UNIT          *MAIN


NAME      TYPE   USE   ATTRIBUTES   REFERENCES

ANS          R   V       SS         9   14
ENORM        R   FN                 9
ERR         EL   V     C SS         2    3    8   10
I            I   V       SS         4    6    7    9
J            I   V       SS         7
X           EI   V       SA1        1    7    9
10                                  4    5    7
1                                   4   13   16
20                                 11   12
2                                  10   14
30                                 14   15


COMMON BLOCKS

ERROR      ERR
```

Figure 2

Key to Figure 3

Note the "S" following the common /ERROR/; this denotes that some variable within the common block has been set by the program unit.

Argument Attributes

column 1:

E explicitly typed
P procedure-name dummy

column 2:

I INTEGER
R REAL
C COMPLEX
D DOUBLE PRECISION
L LOGICAL

column 3:

S value set by program unit

column 4:

S scalar
A array

Global Common Table

column 1:

name of common block

column 2:

S common block initialized by block data subprogram

column 3:

number of COMPLEX and DOUBLE PRECISION entries in common block

column 4:

number of INTEGER, REAL and LOGICAL entries in common block

```
PFORT VERIFIER 12/19/80   VERSION


*** MISMATCHED TYPE IN ARGLIST IN REFERENCE TO --ENORM
*** AT STMT NO     9  IN PGM UNIT *MAIN


ENORM

ARGUMENT ATTRIBUTES      -I-S          -R-A
COMMON BLOCKS            ERROR     S
CALLED BY SUBPROGRAMS    *MAIN
CALLS SUBPROGRAMS        SQRT


*MAIN

COMMON BLOCKS            ERROR     S
CALLS SUBPROGRAMS        ENORM


SQRT

ARGUMENT ATTRIBUTES      -R-S
CALLED BY SUBPROGRAMS    ENORM


COMMON BLOCKS


NAME   SET  DP,COM INT,RL,LOG

ERROR          0         1
```

Figure 3

V. OPTIONS

There are five user-controlled options to activate or suppress various forms of output and verification, and three installation-controlled options to accept or reject certain popular extensions of ANS FORTRAN.

User-Controlled Options

A FORTRAN comment line with an asterisk in column 2 is treated as a control statement by the Verifier; it can be used to activate or suppress various forms of output and verification. The options appear in columns 3-72, separated by commas. The options statement can appear anywhere in the input; the specified options hold for all subsequent input.

The available options with default values italicized are:

L *print a listing of each program unit*
NL do not print a listing

S *print a symbol table for each program unit*
NS do not print a symbol table

R *gather and print out cross references for all symbols for each program unit*
NR do not gather or print cross references

P *perform inter-program-unit checks on this input*
NP do not perform inter-program-unit checks

C compile this program after verification process
NC *do not compile program*

The first three options can be changed as many times as desired; the current value will always be the last value set. The Verifier itself may change the "P" to "NP" if it encounters serious errors in the processing of the input.

Installation-Controlled Options

There are three language extensions that can be accepted or rejected by setting appropriate switches at installation time. (See *Installation of the PFORT Verifier*, Part 2 of this report.)

(1) An optional field of the form "END=label" in the READ statement. If an end-of-file condition is encountered when the READ statement is executed, control is transferred to the label.

(2) The option to specify an array name in a DATA statement, provided that initial values are specified for all elements of the array.

(3) The option to express Hollerith constants as quoted strings. The quote mark is the apostrophe, and a single apostrophe within a quoted string is represented by two consecutive apostrophes.

These extensions are indicated by dotted lines in the graphical syntax charts of Section VI, and are clearly labeled as extensions in the text. Since they are supported by many modern FORTRAN compilers, a program using them may nevertheless be quite portable.

VI. PFORT

This section defines PFORT, the portable subset of ANS FORTRAN. The definition presented here is in four parts. First, we give the graphical syntax for PFORT; this is a restricted version of a full ANS FORTRAN syntax written by M. D. McIlroy [4]. The graphical syntax is followed by supplemental restrictions, numbered according to the syntax rules to which they apply. Second, we describe in detail the features of ANS FORTRAN which are not included in PFORT. Third, we describe the inter-program-unit communications checked by the

Verifier. Fourth, we present the ANS FORTRAN rules not checked by the Verifier.

PFORT: Annotated Syntax

This section consists of three major subsections. The first contains the graphical syntax charts with preliminary interpretive comments. The second consists of notes supplementing the charts. The third contains a cross reference table for all entities in the charts.

The following syntax charts are intended primarily as a reference document for PFORT. 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 embedded hyphens denote syntactic categories. All other characters are literals.

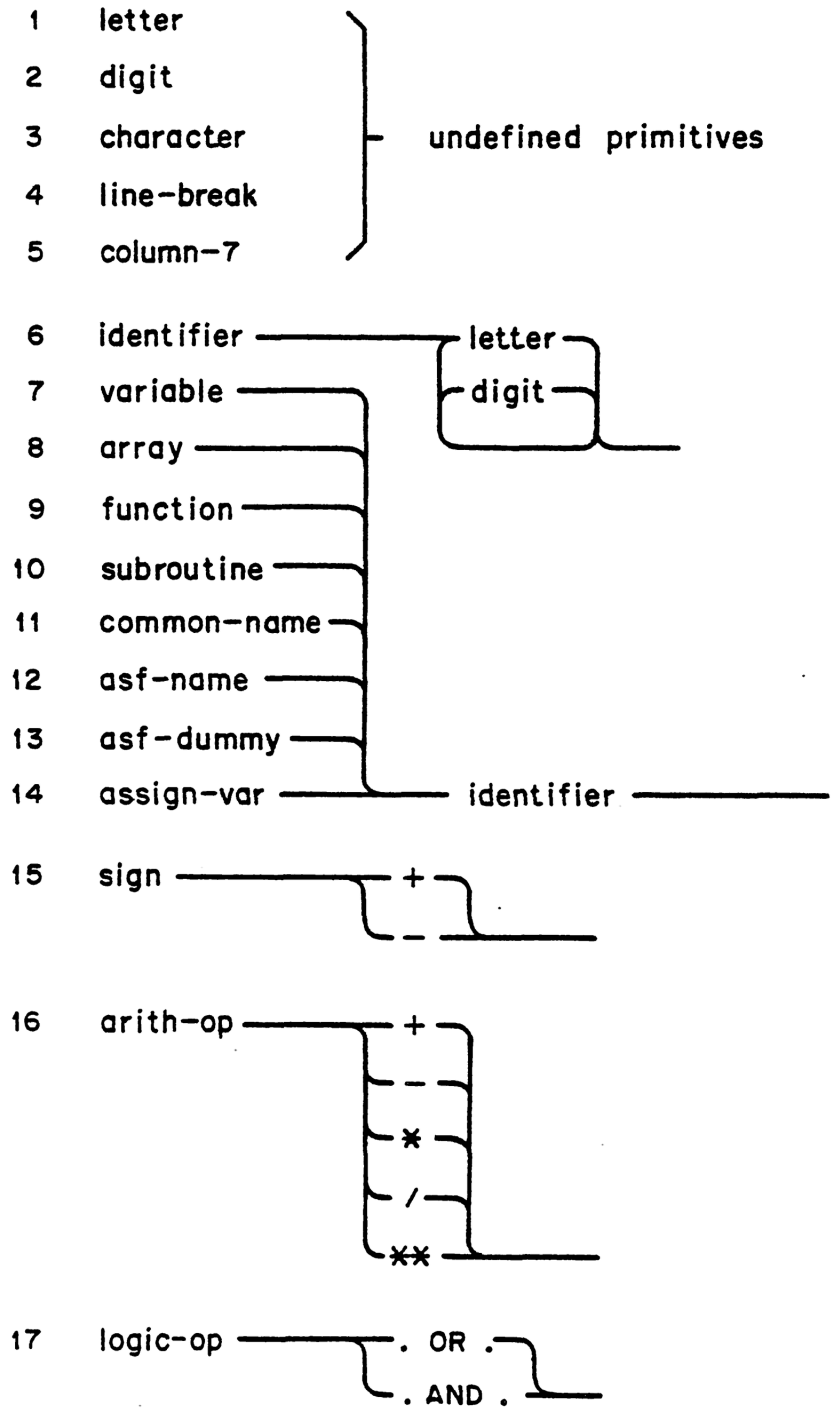
1-5. Primitives

There are several primitive syntactic categories as follows:

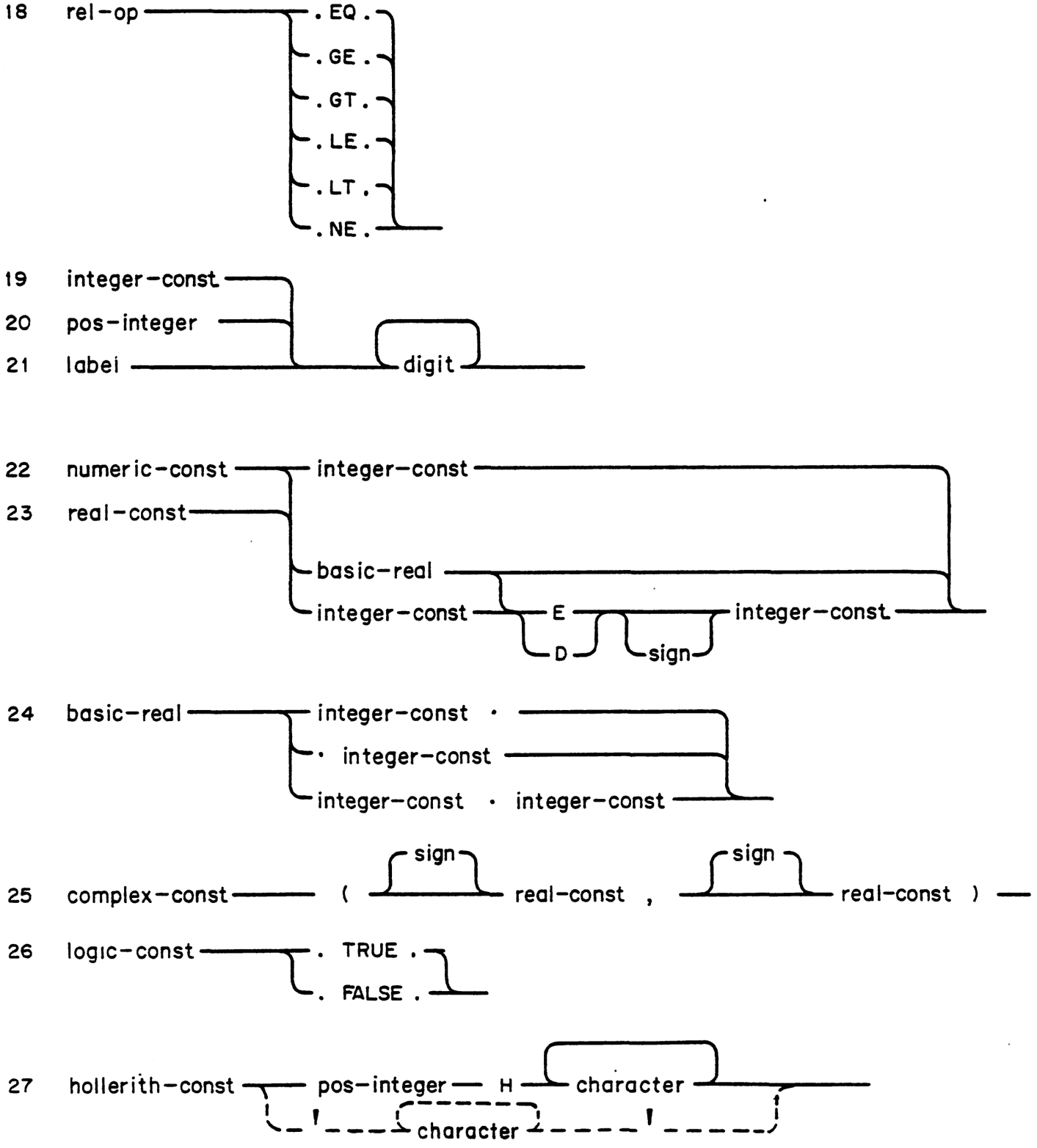
1. letter: any character from the set ABCDEFGHIJKLMNOPQRSTUVWXYZ.
2. digit: any character from the set 0123456789.
3. character: any letter, digit, or character from the set =+-*/(),. and blank; the apostrophe (') is also included if Extension 3 is enabled.
4. line-break: the beginning or end of a FORTRAN initial line with all its continuation lines, or of an end line; comment lines are absorbed into line breaks.
5. column-7: the beginning of a statement proper; column-7 absorbs the blank or 0 in column 6 of the initial line of that statement.

Blanks are not significant unless they appear within a Hollerith constant or a Hollerith field descriptor. Continuation lines and comment lines are not dealt with specifically in this syntax, although the Verifier checks for their correct usage [1, Section 3.4].

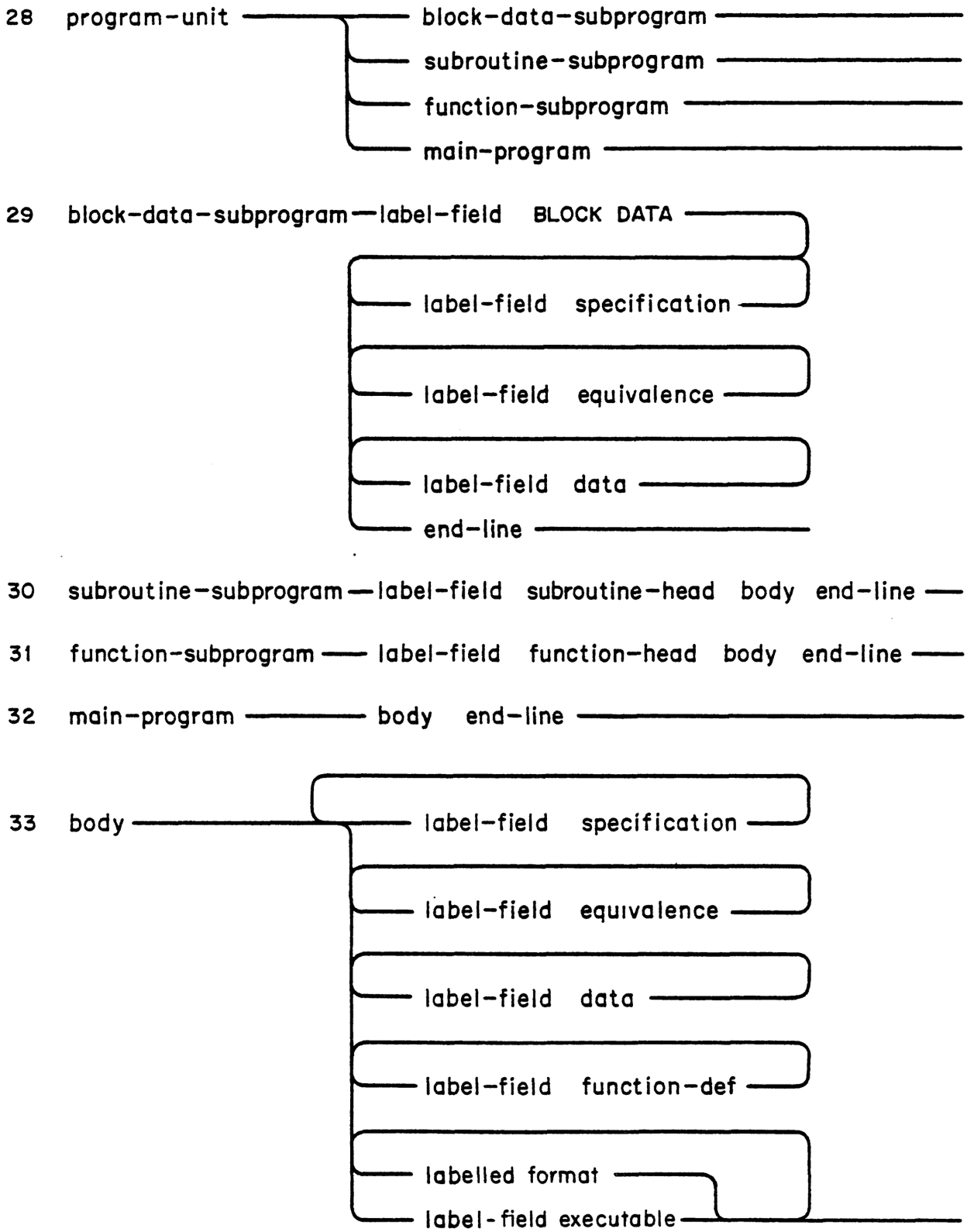
1-17 Primitives , Identifiers , Operators



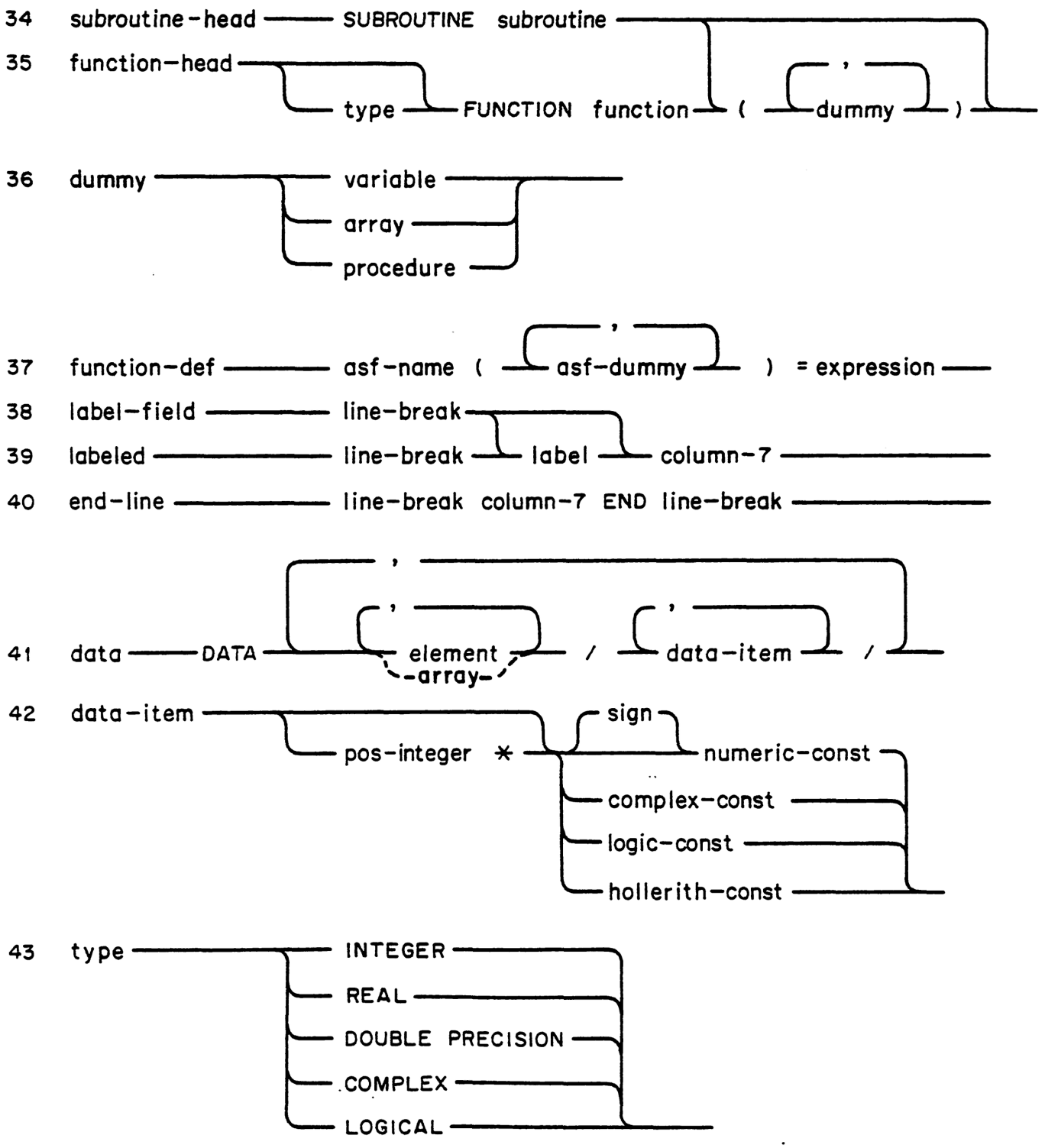
18-27 Operators, Constants



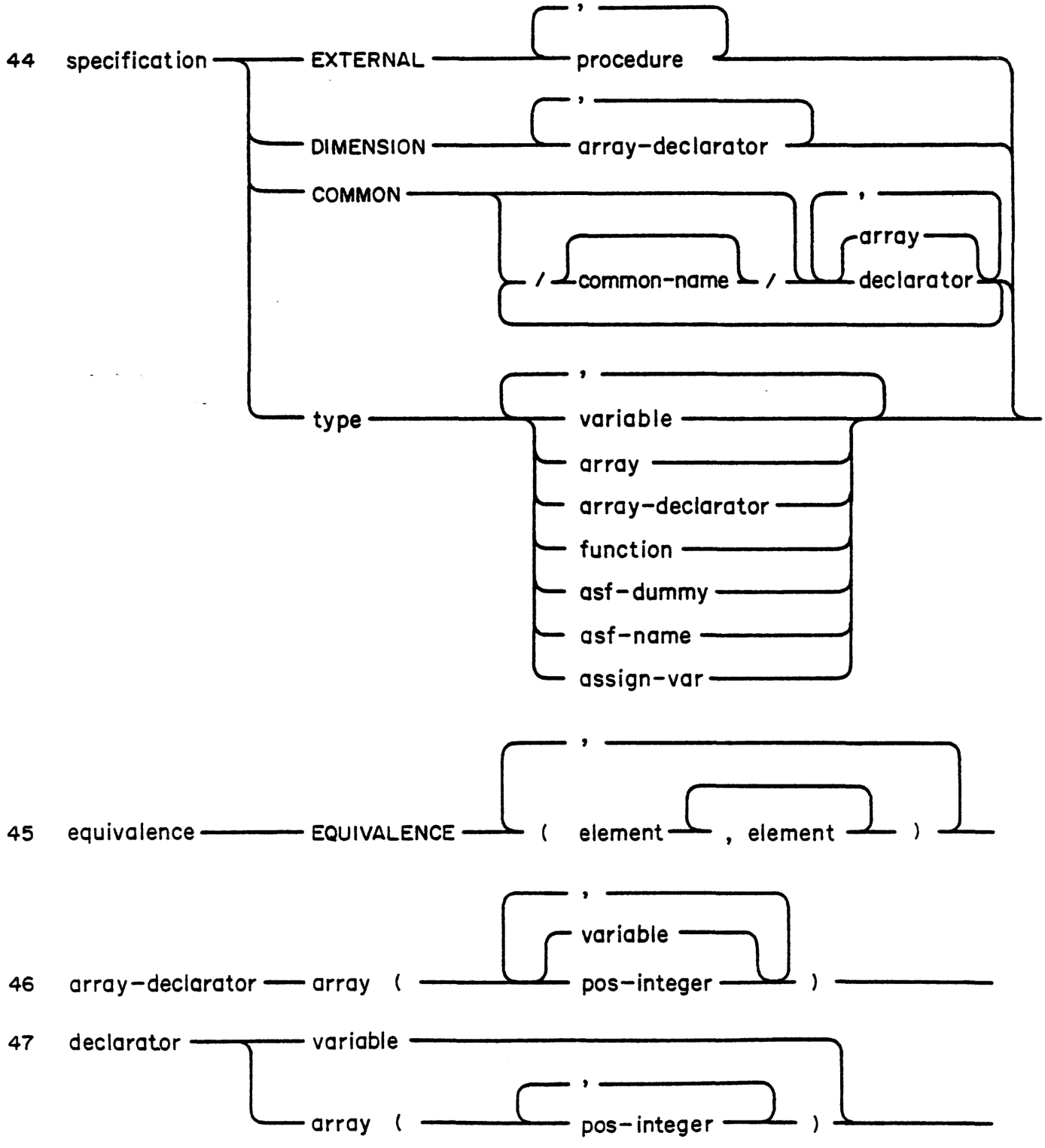
28-33 Program Units



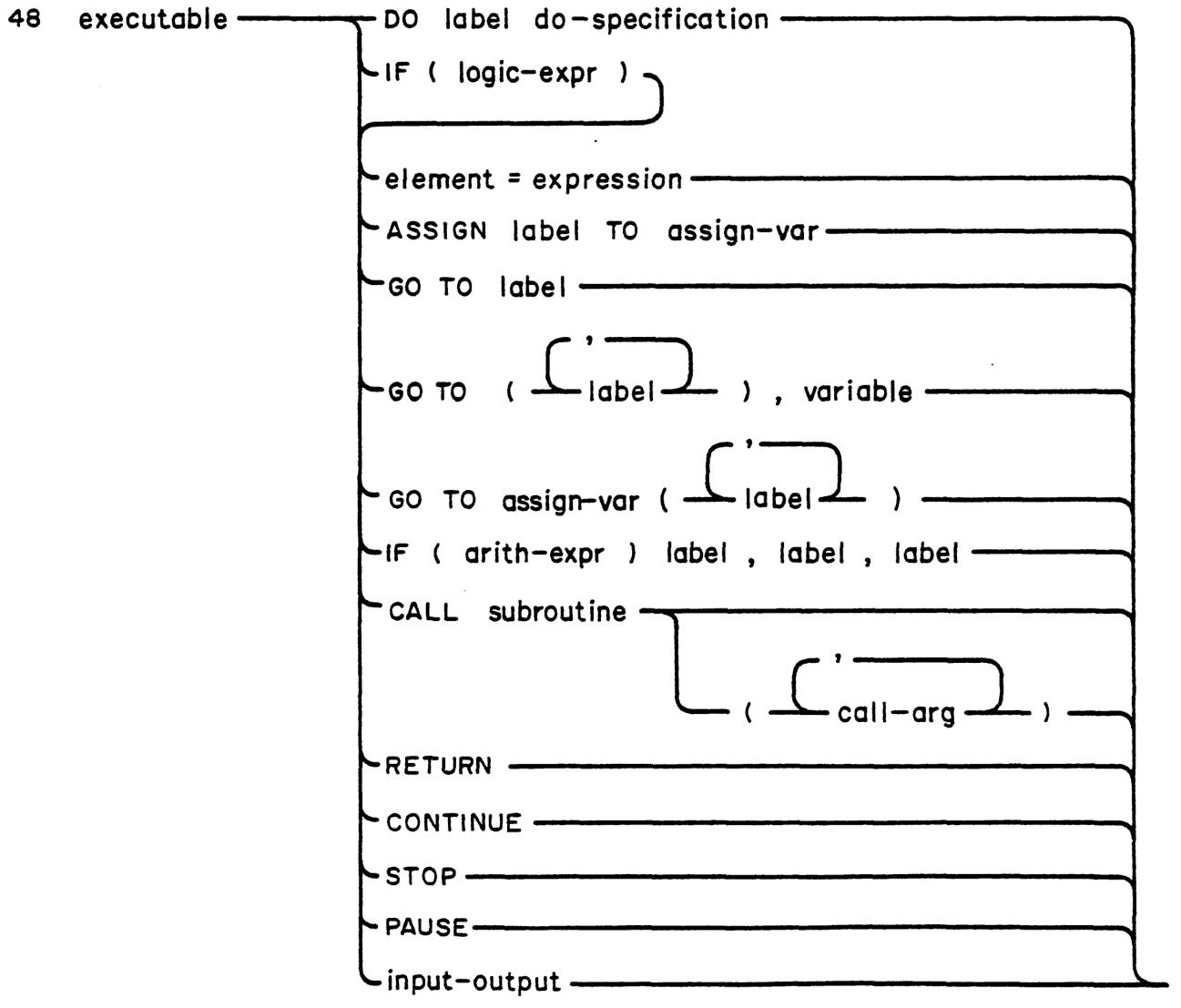
34-43 Program Units, Data Statements, Types



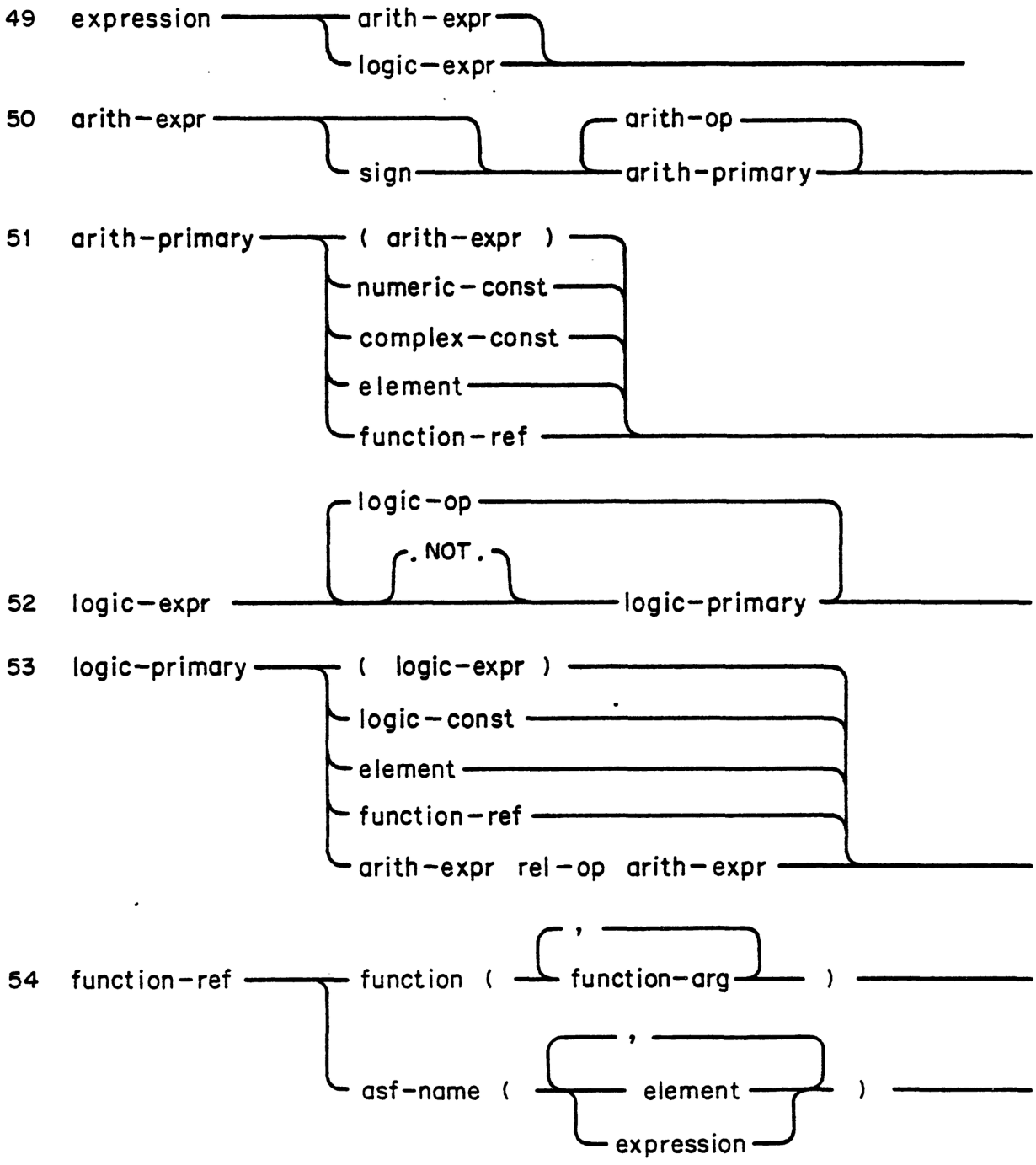
44-47 Specification Statements



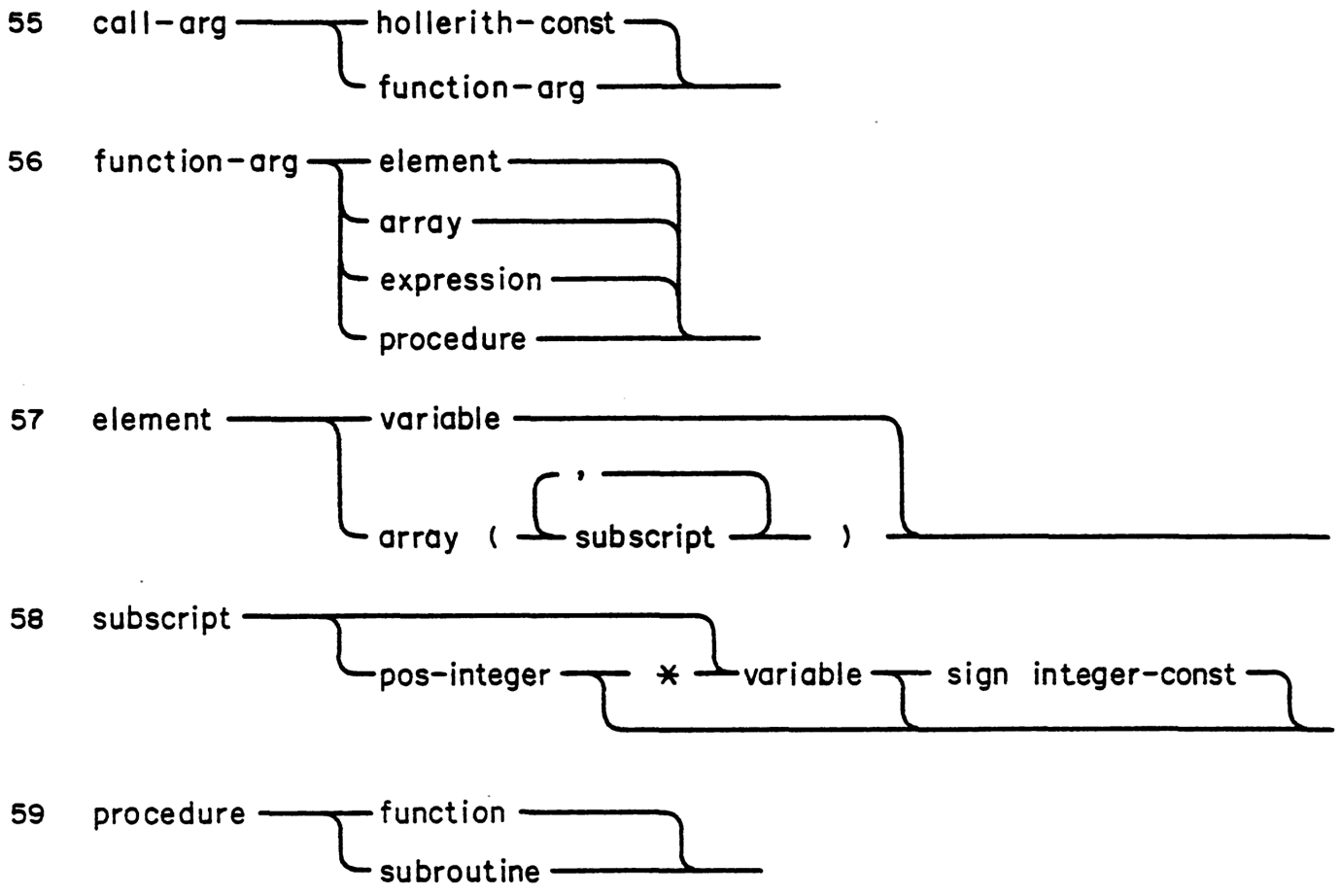
48 Executable Statements



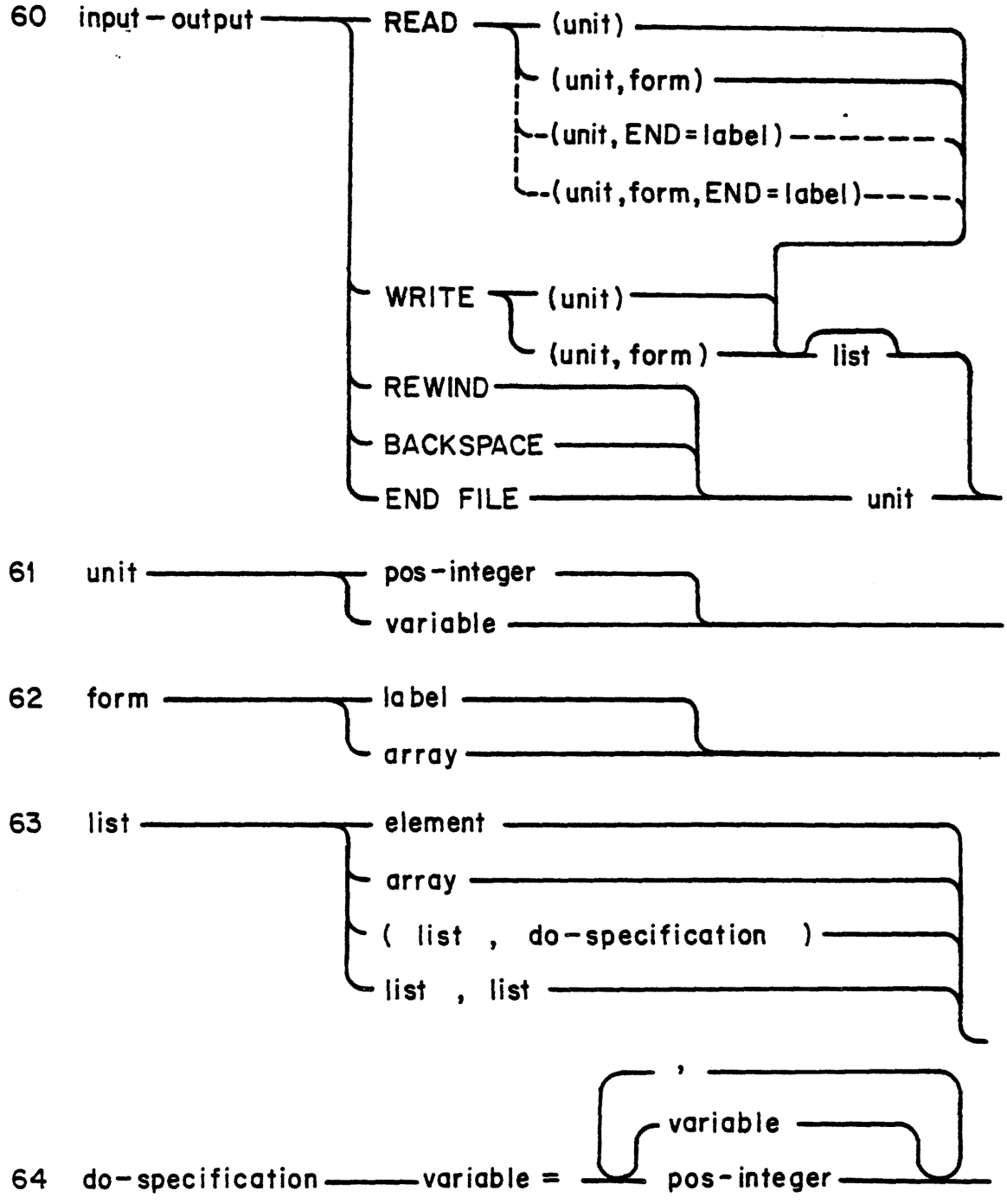
49-54 Expressions



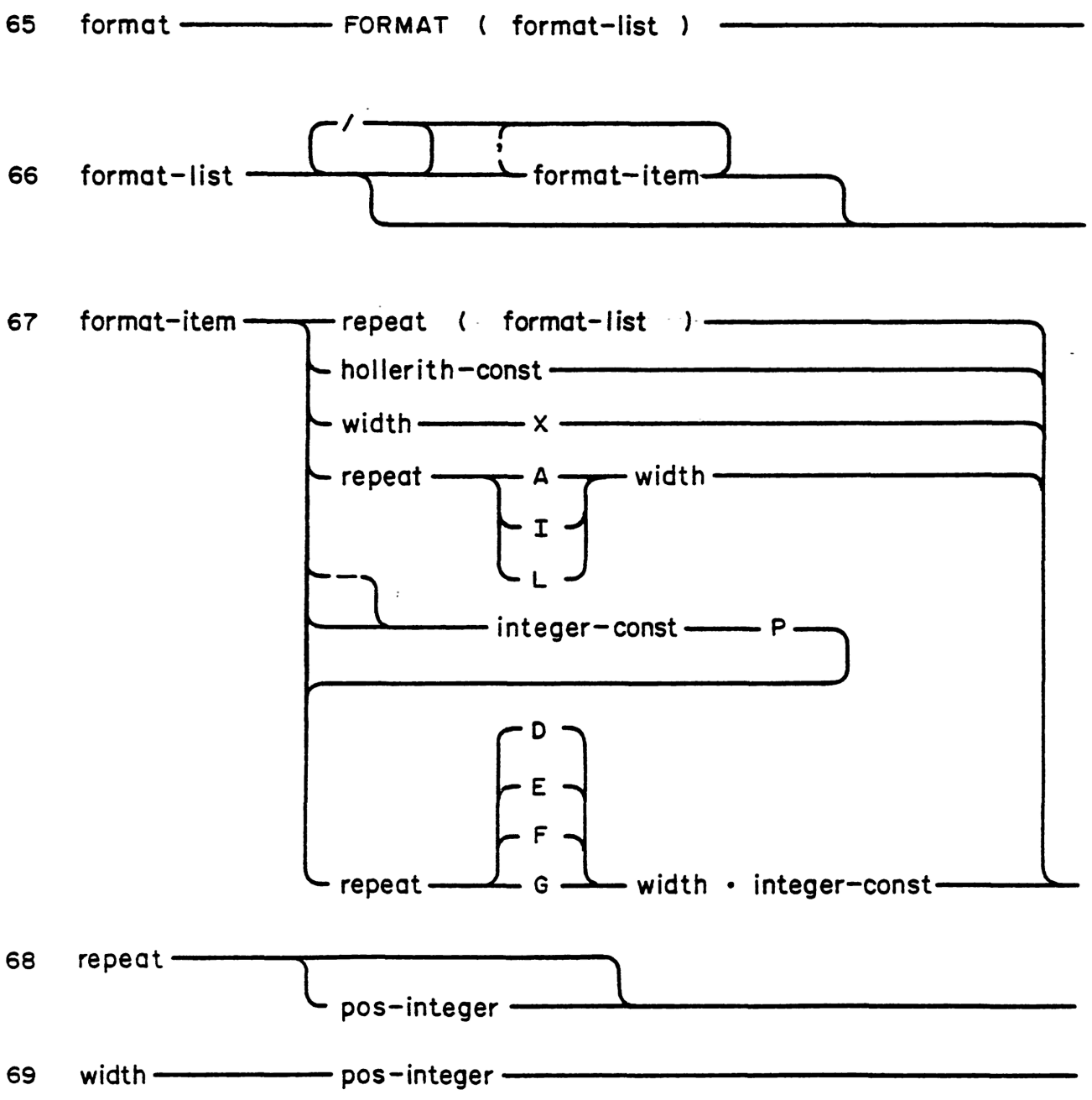
55-59 Arguments, Elements



60-64 Input-Output Statements



65-69 Format Statement



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 6 letters and digits.

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

21. A label contains at most 5 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. When the option of using a quoted string for a Hollerith constant is enabled, a single apostrophe within the string is represented by two consecutive apostrophes. Thus the quoted string 'IT''S' is equivalent to the conventional Hollerith constant 4HIT'S.

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 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. When the option of using an array name within a DATA statement is enabled, the number of data-items supplied for the array must equal the number of its elements.

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

42.* A hollerith-const contains only 1 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 entities must precede all REAL, INTEGER or LOGICAL entities. (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 may not be made equivalent to each other.

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 2 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.* If an array-declarator contains a variable, the type of the variable cannot be declared after this appearance.

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 2 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, DO, arithmetic-IF or logical-IF with one of these as a consequent.

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: there is no "extended range" DO statement 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 meanings 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 reorder the evaluation of the expression as long as parentheses aren't 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:

i. INTEGER op INTEGER, result is INTEGER

ii. REAL op REAL, result is REAL

iii. COMPLEX op COMPLEX, result is COMPLEX

iv. DOUBLE PRECISION op DOUBLE PRECISION, result is DOUBLE PRECISION

v. REAL op DOUBLE PRECISION or DOUBLE PRECISION op REAL, result is DOUBLE PRECISION

vi. 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:

i. INTEGER ** INTEGER, result is INTEGER

ii. REAL ** INTEGER, REAL ** REAL, result is REAL

iii. DOUBLE PRECISION ** INTEGER, DOUBLE PRECISION ** REAL, DOUBLE PRECISION ** DOUBLE PRECISION, REAL ** DOUBLE PRECISION, result is DOUBLE PRECISION

iv. 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:

i. both INTEGER

ii. both REAL

iii. both DOUBLE PRECISION

iv. 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 2 commas.

58. The variable must be of type INTEGER.

60.* The BACKSPACE statement, although within PFORT, is flagged by the Verifier. (It has different meanings 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 1 or 2 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 1 or 2 levels of parentheses.

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

Cross Reference Table

Entry	*Defined*	*Used In*
A		67
arith-expr	50	48,49,51,53
arith-op	16	50
arith-primary	51	50
array	8	36,41,44,46,47,56,57,62,63
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
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		40,60
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	38,39,48,60,62
label-field	38	29,30,31,33
labeled	39	33
letter	1	6
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
.AND.		17
.EQ.		18
.FALSE.		26
.GE.		18
.GT.		18
.LE.		18
.LT.		18
.NE.		18
.NOT.		52
.OR.		17
.TRUE.		26
. (period)		24,67
=		37,48,60,64
+		15,16
-		15,16,67
*		16,42,58
**		16
/		16,41,44,66
' (apostrophe)		27
, (comma)		25,34,35,37,41,44,45,46,47,48, 54,57,60,63,64,66
(...) (parentheses)		25,34,35,37,45,46,47,48,51,53, 54,57,60,63,65,67

PFORT: Differences from ANS FORTRAN

The preceding syntax charts contained notes with asterisks delineating rules in PFORT which are restrictions of ANS FORTRAN. There are some restrictions which cannot be shown in that manner. They are listed here with references to the previous section where applicable.

1. $ is not in the character set (see rule 3).
2. An identifier may not be used within a single program unit for more than one of the purposes listed in rules 7-14.
3. 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:
   i. specification statements, including EQUIVALENCE FORMAT statements, or none of these
   ii. DATA statements, FORMAT statements, arithmetic statement function definitions, or none of these
   iii. DATA statements, FORMAT statements, or none of these, and at least one executable statement
   iv. END statement
4. The octal constant is not in PFORT (see rule 22).
5. EQUIVALENCE statements cannot be intermixed with other specification statements (see rules 29, 33).
6. The number of DOUBLE PRECISION and COMPLEX items must agree in all occurrences of any common block. Note that COMPLEX and DOUBLE PRECISION

items must precede REAL, INTEGER and LOGICAL items (see rule 44).

7. (Dominator Rule). Among all program units containing a particular common block /B/, there must exist a program unit A, such that all nested sequences of subprogram references which reach a program unit containing /B/ also contain A.
8. The extended range DO statement is not in PFORT (see rule 48).
9. Hollerith actual arguments in subprogram calls, must correspond to dummy INTEGER arrays (see rule 55).
10. Redundant parentheses are not permitted in lists in READ and WRITE statements (see rule 63).

PFORT: Inter-program-unit Interfaces and Restrictions

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

1. Common
   a. The total length of a named common block must agree in all occurrences of that block. (As noted above, order and length of data types present must agree.)
2. Subprogram References
   a. The number of actual arguments in a subprogram 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. (As noted above, 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. Note that 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. For examples and further discussion see Section II. We assume a value of an identifier "can be changed" if it appears within the program unit in an executable statement which may change its value if executed.
      i. If an actual argument is an expression or constant and it corresponds to a dummy argument whose value may be changed, it is an unsafe reference.
      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 is changed, it is an unsafe reference. The occurrence of an array name and an array element from the same array in an argument list is considered an 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 [1, Sections 8.3.2, 8.4.2].)
3. Basic External Functions and Intrinsic Functions
   The Verifier recognizes basic external and intrinsic functions [1, 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 external or an 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 [1, 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 [1, 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:

   i. 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 [1, Section 7.1.2.1.3].

   ii. When a DO statement is executed, the variables in the do-specification must be positive and the initial value must be less than the final value [1, Section 7.1.2.8].

   iii. The evaluation of a function reference cannot alter the value of any other primary in the statement in which the reference appears [1, Section 6.4].

   iv. No array element may contain a subscript that during execution, assumes a value less than one or greater than the maximum length specified in the array declarator [1, 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 [1, 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 [1, Section 10.2.8].

5. *Rules involving storage allocation checks*

   The following are examples of such rules:

   i. It is illegal for EQUIVALENCE to result in two different elements of an array being stored in the same location [1, Section 7.2.1.4].

   ii. A storage unit may only be initialized once in an executable program [1, Section 10.1.3].

   iii. The use of variables of different types which share the same storage location through use of EQUIVALENCE is restricted by the Standard [1, Section 10.2].

6. *Rules concerning the compatibility of I/O list elements with their corresponding format fields and those concerning I/O conversions [1, Sections 7.2.3.4, 7.2.3].*

VII. ACKNOWLEDGMENTS

I am most grateful to A. D. Hall, Jr. for his contributions to this project. I also am grateful to W. S. Brown and M. D. McIlroy.

VIII. REFERENCES

1. "American National Standard FORTRAN," American National Standards Institute, New York, New York (1966). (See also "FORTRAN vs. Basic FORTRAN - A Programming Language for Information Processing on Automatic Data Processing Systems," Communications of the ACM, Vol. 7, October 1964, pp. 591-624.)

2. "Clarifications of FORTRAN Standards-Initial Progress," Communications of the ACM, Vol. 12, May 1969, pp. 289-294.

3. "Clarifications of FORTRAN Standards - Second Report," Communications of the ACM, Vol. 14, October 1971, pp. 628-642.

4. M. D. McIlroy, "ANS FORTRAN Charts," Computing Science Technical Report No. 13, Bell Laboratories, Murray Hill, New Jersey, March 1974. Also pp. 1-15 in *Using ANS FORTRAN*, ed. Gordon Lyon, National Bureau of Standards, Washington, D.C., March 1980.

5. B. G. Ryder, "The PFORT Verifier," Software Practice and Experience, Vol. 4, No. 4, October 1974, pp. 359-377.

6. A. D. Hall and B. G. Ryder, "Installation of the PFORT Verifier," Bell Laboratories, Murray Hill, New Jersey, 1973. (Part 2 of this report.)

Installation of the PFORT Verifier

A. D. Hall and B. G. Ryder
Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

The PFORT Verifier is a program which checks other FORTRAN programs for conformance to PFORT, a portable subset of American National Standard FORTRAN. The Verifier is itself written in PFORT and can therefore easily be installed on any sufficiently large computer with a compiler for ANS FORTRAN.

This paper gives the information needed to install the Verifier on most computers and outlines certain machine and installation dependent enhancements that can improve its performance and utility.

Installation of the PFORT Verifier

A. D. Hall and B. G. Ryder
Bell Laboratories
Murray Hill, New Jersey 07974

I. INTRODUCTION

The PFORT Verifier [1,2] is a program which checks other FORTRAN programs for conformance to PFORT, a portable subset of American National Standard FORTRAN [3-5]. The Verifier is itself written in PFORT and can therefore easily be installed on any computer with sufficient memory and a compiler for ANS FORTRAN.

Section II of this paper gives most of the information needed to install the Verifier on most computers. Section III outlines certain enhancements that can improve its performance and utility.

II. INITIAL INSTALLATION

To install the Verifier, one must write two short assembly language subroutines for packing and unpacking character data and if necessary modify a few assignment statements in the MAIN program. Once this has been done, the Verifier can be compiled and tested.

Outlines of the two assembly language primitives are given below in an informal FORTRAN-like language using comment statements where FORTRAN coding would have been excessively clumsy. Listings of these primitives for several machines appear in the Appendix.

```
      SUBROUTINE S5PACK(A,B,N)

      INTEGER A(*),B(*),N

C     N > 0

C     A(1), A(2), ..., A(N) contain characters in
C     A1 format. S5PACK packs these characters
C     into the elements of the array B in An for-
C     mat  (n = NOCHAR, see below), using only as
C     many elements of B as necessary.

C     The last element of B used must be padded
C     with trailing blanks. Unused elements of B
C     are not affected.

      RETURN
      END
```

```
      SUBROUTINE S5UNPK(A,B,N)

      INTEGER A(*),B(*),N

C     N > 0

C     A(1), A(2), ..., contain characters in An
C     format (n = NOCHAR, see below). S5UNPK un-
C     packs the first N characters, putting them
C     into B(1), B(2), ..., B(N) in A1 format.

C     Each element of B used must be padded with
C     trailing blanks. Unused elements of B are
C     not affected.

      RETURN
      END
```

The variables that must be assigned correct values in the MAIN program (sequenced VFBA in cols. 73-76) are described below:

NOCHAR — This must be set to the number of characters that can be packed into a FORTRAN integer. In the distributed version of the Verifier, this is set to 6.

INUT — This is the logical unit number used for Verifier input. In the distributed version of the Verifier, this is set to 5.

OUTUT — This is the logical unit number used for Verifier output. In the distributed version of the Verifier, this is set to 6.

OUTUT2

OUTUT3

OUTUT4 — These logical units are used by the Verifier for holding temporary data. Input and output operations are performed using unformatted (binary) READ and WRITE statements. In the distributed version of the Verifier, these units are set to 12, 13, and 14, respectively.

III. REFINEMENTS

In this section we describe a number of refinements that can be made to the Verifier in order to improve its utility and performance. Whether or not these changes are worth the effort involved will depend on the intended use of the Verifier.

III.1 End-of-input Detection

In the distributed version of the Verifier, "end-of-input" is signalled by a record (card) containing a "." (dot) in the first character position [2, p. 10]. This restriction can be removed by changing a single READ statement in the function IN (labeled VFLK) as described in the comments therein.

III.2 The COMPILE Option

The Verifier is designed so that it can easily be used as a preprocessor for the local FORTRAN compiler. This is accomplished by providing a locally written version of the subroutine COMPIL which is invoked by the Verifier after processing is complete. The default version (sequenced VFBF) simply returns.

COMPIL has a single LOGICAL argument which is .TRUE. if and only if the "COMPILE" option [2, p. 17] has been used and no major errors have been found during verification.

A typical version of COMPIL will perform the following steps:

1. Rewind and close the Verifier input unit (INUT) and close the Verifier output unit (OUTUT).
2. If the argument is .FALSE., signal "compilation failure" to the host operating system and RETURN.
3. If the argument is .TRUE., call (overlay) the local FORTRAN compiler and pass to it the input processed by the Verifier.

III.3 Overlays

In operating systems where core memory is a critical resource, the size of the Verifier may be reduced through the use of overlays. However, exchanging core memory utilization for input-output channel utilization is not necessarily economical and may adversely effect the performance and cost.

The following picture illustrates how the Verifier can be organized into a resident portion and three overlays which are brought in as needed.

Resident Subprograms

BLOCK DATA (4)
MAIN program
OVRLAY
SATT1
IGATT1
EXCH
SSORT
ERROR1
ERROR2
S5PACK
S5UNPK
FORTRAN Library

Overlay 1		*Overlay 2*		*Overlay 3*
PU	TYPST2	CHECKS	OUT2	COMPIL
ASSASF	CALLS	COMCHK	OUTCOM	
ASSIGN	DATA	INSYM	OUTLAT	
COMMON	GETTOK	SCAN	OUT2A	
ARDECL	TOKCOM	SATT2	SETNOD	
DIMENS	TOKLOG	IGATT2	MKCOM	
DOCHK	TOKLOP	MATCH	SETARG	
DOVAR	TOKRL	UNSAFE	SETASF	
EQUIV	DOSTMT	FINDND	SETCOM	
EXTERN	DOSPEC	COMPAR	CHK1	
FIXLAB	END	FINDCM	CHK2	
FORMAT	OUTSYM	INREF	CHK3	
GOTO	RDLIST	CONSTR	ASLEV	
IFS	RFLIST	OUT2C	FIND	
INTEXT	SORT	SETEXT	INVOKE	
IO	EXPR	SETPD	PROC	
NEXTOK	POP	SETREF	RDSYM	
SEPAR	INSTMT		CHKALL	
SETNAM	IN			
SUBS	INOPT			
SUBFCN	LIST			
TOKLAB	IDLIST			
TOKPNO	ID			
TYPE	LDOVAR			
TYPST	LOOKUP			
	MAPCHR			

In those operating systems which do not automatically load an overlay when a call is made to one of the subprograms it contains, the subroutine OVRLAY can be rewritten to make the necessary operating system call. This subroutine is called by the Verifier prior to a call to a subroutine which is in an overlay other than the one currently being executed. OVRLAY has a single INTEGER argument indicating which of the above overlays is to be loaded. The default version (sequenced VFBK) simply returns.

III.4 Critical Subprograms

The distributed version of the Verifier maps characters into an internal code by linearly searching a table to find a matching character. The index of the matching character is then used to look up the desired internal code from another table. Obviously, this is an inefficient technique, but necessary if a program of this nature is to be portable.

The processor time consumed by the Verifier can be substantially reduced by recoding the function MAPCHR (labeled VFMF) in assembly language. This version of MAPCHR can use the character directly as an index into a table of internal codes. Care should be taken to ensure that such a table embodies the same mapping as implemented by the distributed version of MAPCHR.

III.5 Internal Table Overflow

To simplify the implementation of the Verifier, only rudimentary forms of storage allocation are used. Consequently there are a number of internal tables that might overflow when a large program or subprogram is verified. Typically this is indicated by a diagnostic message of the form:

In name TABLE OVERFLOW OF table

where *name* is the name of the subprogram producing the diagnostic and *table* is the name of the array that overflowed. In all cases, the table in question resides in a named COMMON region and its size may be increased simply by increasing the size of the region and setting the length parameter to the appropriate value in the MAIN program.

IV. Extensions

There are three extensions to PFORT which can be recognized by the Verifier if the appropriate switches are set in the main program before it is compiled. These extensions are described in Section V of the first part of this report, "The PFORT Verifier: User's Guide". The labeled COMMON block SWS contains a logical array SW whose element values are set in the main program to control the recognition and analysis of these extensions.

SW(1) - normally set .FALSE., should be set .TRUE. on line VFBA1720 in the main program if the "END=" option is to be permitted in READ statements.

SW(2) - normally set .FALSE., should be set .TRUE. on line VFBA1721 in the main program if the quoted string form of hollerith constants is to be permitted.

SW(3) - normally set .FALSE., should be set .TRUE. on line VFBA1722 in the main program if array names are to be permitted in DATA statements.

V. DIFFICULTIES

Any difficulties should be reported to:

C. D. McLaughlin
Bell Laboratories
Whippany Rd.
Whippany, N.J. 07981

References

1. B. G. Ryder, "The PFORT Verifier," Software Practice and Experience, Vol. 4, No. 4, October-December 1974, pp. 359-377.
2. B. G. Ryder, "The PFORT Verifier - User's Guide," Computer Science Technical Report #12, Bell Laboratories, Murray Hill, New Jersey, March 1973, revised March 1975.
3. American National Standard FORTRAN, American National Standards Institute, New York, N.Y., 1966.
4. "Clarifications of FORTRAN Standards - Initial Progress," Communications of the ACM, Vol. 12, 1969, pp. 289-294.
5. "Clarifications of FORTRAN Standards - Second Report," Communications of the ACM, Vol. 14, 1971, pp. 628-642.

Appendix - Primitives

A.1 Honeywell 6000 Series

```
            TTL     S5PACK AND S5UNPK
            LBL     PACKER
            REM
            SYMDEF  S5PACK
            REM
S5PACK      EAQ     3,1*        GET OUTPUT ADDRESS
            STQ     TALLY       TO FORM TALLY
            LDQ     4,1*        FROM CHARACTER COUNT
            QLS     6
            QRSQ    TALLY
            EAX0    2,1*        GET INPUT ADDRESS
NXPA        LDQ     0,0         GET CHARACTER
            LLS     6
            STA     TALLY,SC    PACK IT AWAY
            ADLX0   1,DU        BUMP INPUT POINTER
            TTF     NXPA        AND GET ANOTHER
            REM
            LDA     =6H         GET BLANKS
            SZN     TP,CI       SENSE CHARACTER POSITION
            TZE     0,1*        IF ZERO, DONE
            STA     TALLY,SC    PAD ONCE
            TRA     *-3         AND TRY AGAIN
*
            SYMDEF  S5UNPK
            REM
S5UNPK      EAQ     2,1*        GET INPUT ADDRESS
            STQ     TALLY       AND FORM TALLY
            LDQ     4,1*        FROM CHARACTER COUNT
            QLS     6
            ORSQ    TALLY
            EAX0    3,1*        GET OUTPUT ADDRESS
NXUN        LDA     TALLY,SC    GET CHARACTER
            LDQ     =6H         GET BLANK PADDING
            LRL     6           FORM UNPACKED CHARACTER
            STQ     0,0         AND STORE IT
            ADLX0   1,DU        BUMP OUTPUT POINTER
            TTF     NXUN        AND TRY AGAIN
            TRA     0,1*        DONE
            REM
TALLY       ZERO
TP          TALLY   TALLY,0,5
            END
```

Appendix - Primitives

A.2 IBM 360/370 Series

```
S5PRIM  CSECT
        ENTRY   S5UNPK
        ENTRY   S5PACK
        SPACE   5
        CNOP    0,4
S5PACK  EQU     *
        USING   *,15
        STM     2,5,28(13)
        LM      2,4,0(1)
        L       4,0(4)
        LR      5,4
        S       5,=F'1'
        SRL     5,2
        SLL     5,2
        LA      5,0(5,3)
        MVC     1(3,5),=CL3'  '
PKTOP   MVC     0(1,3),0(2)
        LA      3,1(3)
        LA      2,4(2)
        BCT     4,PKTOP
        LM      2,5,28(13)
        BR      14
        DROP    15
        SPACE   5
        CNOP    0,4
S5UNPK  EQU     *
        USING   *,15
        STM     2,4,28(13)
        LM      2,4,0(1)
        L       4,0(4)
UNTOP   MVC     0(1,3),0(2)
        MVC     1(3,3),=CL3'  '
        LA      2,1(2)
        LA      3,4(3)
        BCT     4,UNTOP
        LM      2,4,28(13)
        BR      14
        DROP    15
        END
```

Appendix - Primitives

A.3 CDC 6000/7000 Series (FTN Compiler)

```
            IDENT   S5UNPK
            ENTRY   S5UNPK          CALL S5UNPK(A,B,N)
            VFD     36/0HS5UNPK,24/3
S5UNPK      DATA    0
            SX6     A0
            SA6     SAVEA0          SAVE A0 REGISTER
            SB7     1
            SB1     X1              ADRESS OF A
            SA1     A1+B7
            SB2     X1
            SA1     A1+B7
            SB3     X1              ADRESS OF N
            SA5     B3              N = NUMBER OF CHARS
            SA1     B1              A(1) = FIRST WORD
            SA2     BLNKS           9 BLANKS
            SB1     1
            MX0     6               CHAR MASK
            SB5     10              LOOP INDEX
            SB3     X5
LOOP        BX7     X0*X1
            LX1     6
            SB5     B5-B1
            BX7     X7+X2           PAD WITH BLANKS
            SB3     B3-B1
            SA7     B2
            LE      B3,B0,S5UNP1
            SB2     B2+B1
            GT      B5,B0,LOOP
            SA1     A1+B1           NEXT WORD
            SB5     10
            EQ      LOOP
S5UNP1      SA3     SAVEA0
            SA0     X3
            EQ      S5UNPK
BLNKS       DATA    9R
SAVEA0      BSSZ    1
            END
```

```
         IDENT   S5PACK
         ENTRY   S5PACK          CALL S5PACK(A,B,N)
         VFD     36/0HS5PACK,24/3
S5PACK   DATA    0
         SX6     A0
         SA6     SAVEA0         SAVE A0 REGISTER
         SB7     1
         SB1     X1             ADRESS OF A
         SA1     A1+B7
         SB2     X1
         SA1     A1+B7
         SB3     X1             ADRESS OF N
         SA5     B3             N = NUMBER OF CHARS
         SA1     B1             A(1) = FIRST CHAR
         SB1     1
         SB6     6
         MX0     6
         SB3     X5             INITIALIZE LOOP INDEX
         SB4     B0
         MX6     0
         SB7     60
LOOP     BX7     X0*X1
         SA1     A1+B1          A(I)
         LX7     B7,X7          POSITION CHAR
         SB7     B7-B6
         SB3     B3-B1
         BX6     X6+X7          ADD CHAR
         EQ      B3,B0,OUT
         GT      B7,B0,LOOP
         SA6     B2+B4          STORE IN B(J)
         SB4     B4+B1
         SB7     60
         MX6     0
         EQ      LOOP
OUT      SA1     A1-B1          FETCH LAST CHAR AND USE
         SB3     59             TRAILING BLANKS
         MX0     1
         SB7     B3-B7          BITS ALREADY FILLED -1
         AX0     B7,X0
         BX7     -X0*X1
         BX6     X6+X7
         SA6     B2+B4
         SA3     SAVEA0
         SA0     X3
         EQ      S5PACK
SAVEA0   BSSZ    1
         END
```

Appendix - Primitives

A.4 CDC 6000/7000 (RUN Compiler)

```
            IDENT   S5PACK
            ENTRY   S5PACK          CALL S5PACK(A,B,N)
            VFD     36/0HS5PACK,24/3
S5PACK      DATA    0
            SA5     B3              N = NUMBER OF CHARS
            SA1     B1              A(1) = FIRST CHAR
            SB1     1
            SB6     6
            MX0     6
            SB3     X5              INITIALIZE LOOP INDEX
            SB4     B0
            MX6     0
            SB7     60
LOOP        BX7     X0*X1
            SA1     A1+B1           A(I)
            LX7     B7,X7           POSITION CHAR
            SB7     B7-B6
            SB3     B3-B1
            BX6     X6+X7           ADD CHAR
            EQ      B3,B0,OUT
            GT      B7,B0,LOOP
            SA6     B2+B4           STORE IN B(J)
            SB4     B4+B1
            SB7     60
            MX6     0
            EQ      LOOP
OUT         SA1     A1-B1           FETCH LAST CHAR AND USE
            SB3     59              TRAILING BLANKS
            MX0     1
            SB7     B3-B7           BITS ALREADY FILLED - 1
            AX0     B7,X0
            BX7     -X0*X1
            BX6     X6+X7
            SA6     B2+B4
            EQ      S5PACK
            END
```

Appendix - Primitives

```
         IDENT   S5UNPK
         ENTRY   S5UNPK              CALL S5UNPK(A,B,N)
         VFD     36/0HS5UNPK,24/3
S5UNPK   DATA    0
         SA5     B3                 N = NUMBER OF CHARS
         SA1     B1                 A(1) = FIRST WORD
         SA2     BLNKS              9 BLANKS
         SB1     1
         MX0     6                  CHAR MASK
         SB5     10                 LOOP INDEX
         SB3     X5
LOOP     BX7     X0*X1
         LX1     6
         SB5     B5-B1
         BX7     X7+X2              PAD WITH BLANKS
         SB3     B3-B1
         SA7     B2
         LE      B3,B0,S5UNPK
         SB2     B2+B1
         GT      B5,B0,LOOP
         SA1     A1+B1              NEXT WORD
         SB5     10
         EQ      LOOP
BLNKS    DATA    9R
         END
```

Appendix - Primitives

A.5 DEC PDP-10 Series

```
            TITLE   S5
            ENTRY   S5PACK,S5UNPK
A=3
B=4
N=5
P=6
S5PACK:     JSR     SAVE
            MOVE    P,[POINT    7,]
            HRR     P,B
            JUMPLE  N,E
            MOVN    N,N
            HRLZ    N,N
            HRR     N,A
L1:         LDB     BP
            IDPB    P
            AOBJN   N,L1
E:          JRST    RET
BP:         POINT   7,(N),6

S5UNPK:     JSR     SAVE
            MOVE    1,[ASCII/   /]
            MOVE    P,[POINT    7,1]
            HRR     P,A
            MOVN    N,N
            HRLZ    N,N
            HRR     N,B
L2:         MOVEM   1,0(N)
            ILDB    P
            DPB     BP
            AOBJN   N,L2
RET:        MOVE    A,AS
            MOVE    B,BS
            MOVE    N,NS
            MOVE    P,PS
            POPJ    17,

SAVE:       0
            MOVEM   A,AS
            MOVEM   B,BS
            MOVEM   N,NS
            MOVEM   P,PS
            MOVEI   A,@0(16)
            MOVEI   B,@1(16)
            MOVE    N,@2(16)
            JRST    @SAVE
AS:         XWD
BS:         XWD
NS:         XWD
PS:         XWD
            END
```

Appendix - Primitives

A.6 Univac 1100 Series

```
      SUBROUTINE S5PACK(A,B,N)
C
C     PACK CHARACTER STRINGS
C
      INTEGER A(1),B(1)
C
      ENCODE(6,5,B) (A(I), I=1,N)
    5 FORMAT(6A1)
C
      RETURN
C
      END


      SUBROUTINE S5UNPK(A,B,N)
C
C     UNPACK CHARACTER STRINGS
C
      INTEGER A(1),B(1)
C
      DECODE(6,5,A) (B(I), I=1,N)
    5 FORMAT(6A1)
C
      RETURN
C
      END
```