ANSI X3.9-1978
ISO 1539-1980 (E)

American National Standard

Adopted for Use by the Federal Government

FIPS

FIPS PUB 69-1
See Notice on Inside Front Cover

adopted by the
DEPARTMENT OF DEFENSE
see acceptance notice on inside front cover

programming language FORTRAN

ANSI X3.9-1978

american national standards institute, inc.
1430 broadway, new york, new york 10018

FIPS

This standard has been adopted for Federal Government use.

Details concerning its use within the Federal Government are contained in FIPS PUB 69, FORTRAN. For a complete list of the publications available in the Federal Information Processing Standards Series, write to the Office of Technical Information and Publications, National Bureau of Standards, Washington, D.C. 20234.

DOD

ANSI X3.9-1978
15 November 1978

ACCEPTANCE NOTICE

The above Industry Standardization Document was adopted 15 November 1978 and is approved for voluntary use by the Departments of the Army, Navy and Air Force. The indicated industry groups have forwarded the clearances required by existing regulations. Copies of the document are stocked by DOD Single Stock Point, U.S. Naval Publications and Forms Center, Philadelphia, Pa., for issue to DOD activities only.

Title of Document:	American National Standard Programming Language FORTRAN
Date of Specific Issue Adopted:	3 April 1978
Releasing Industry Group:	American National Standards Institute

Custodians:
Army — AD
Navy — OM
Air Force — 02

Military Coordinating Activity
Air Force — 02

Project Number IPSC — 0052

ISO

This standard has been adopted as ISO International Standard ISO 1539-1980(E). ISO (the International Organization for Standardization) is a worldwide federation of national standards institutes (ISO member bodies). The work of developing International Standards is carried out through ISO technical committees. Every member body interested in a subject for which a technical committee has been set up has the right to be represented on that committee. International organizations, governmental and non-governmental, in liaison with ISO, also take part in the work.

Draft International Standards adopted by the technical committees are circulated to the member bodies for approval before their acceptance as International Standards by the ISO Council.

International Standard ISO 1539 was developed by Technical Committee ISO/TC 97, Computers and information processing, and was circulated to the member bodies in March 1979.

It has been approved by the member bodies of the following countries:

Belgium	Germany, F. R.	Netherlands	Sweden
Canada	Hungary	Poland	Switzerland
Czechoslovakia	Italy	Romania	United Kingdom
Egypt, Arab Rep. of	Japan	South Africa, Rep. of	USA
Finland	Libyan Arab Jamahiriya	Spain	USSR
France			

No member body expressed disapproval of the document.

This International Standard cancels and replaces ISO Recommendation R 1539-1972, of which it constitutes a technical revision.

ANSI®
X3.9-1978
Revision of
ANSI X3.9-1966

American National Standard Programming Language FORTRAN

Secretariat

Computer and Business Equipment Manufacturers Association

Approved April 3, 1978

American National Standards Institute, Inc

American National Standard

An American National Standard implies a consensus of those substantially concerned with its scope and provisions. An American National Standard is intended as a guide to aid the manufacturer, the consumer, and the general public. The existence of an American National Standard does not in any respect preclude anyone, whether he has approved the standard or not, from manufacturing, marketing, purchasing, or using products, processes, or procedures not conforming to the standard. American National Standards are subject to periodic review and users are cautioned to obtain the latest editions.

CAUTION NOTICE: This American National Standard may be revised or withdrawn at any time. The procedures of the American National Standards Institute require that action be taken to reaffirm, revise, or withdraw this standard no later than five years from the date of publication. Purchasers of American National Standards may receive current information on all standards by calling or writing the American National Standards Institute.

Published by

American National Standards Institute
1430 Broadway, New York, New York 10018

Printed in the United States of America

P4.5M1179/1650

Foreword

(This Foreword is not a part of American National Standard Programming Language FORTRAN, ANSI X3.9-1978.)

American National Standard Programming Language FORTRAN, ANSI X3.9-1978, specifies the form and establishes the interpretation of programs expressed in the FORTRAN language. It consists of a full language and a subset language. Its purpose is to promote portability of FORTRAN programs for use on a variety of data processing systems.

It is suggested that the designation FORTRAN 77 be used to distinguish this standard from previous FORTRAN standards and any possible future revisions.

FORTRAN 77 is a revision of American National Standard FORTRAN, ANSI X3.9-1966. It describes two levels of the FORTRAN language, referred to as FORTRAN and Subset FORTRAN. FORTRAN is the full language and appears on the righthand pages; Subset FORTRAN is a subset of the full language and appears on the lefthand pages. Because FORTRAN 77 includes the subset, American National Standard Basic FORTRAN, ANSI X3.10-1966, has been withdrawn.

This standard was approved as an American National Standard by the American National Standards Institute on April 3, 1978.

Suggestions for improvement of this standard will be welcome. They should be sent to the American National Standards Institute, 1430 Broadway, New York, N.Y. 10018.

This standard was processed and approved for submittal to ANSI by the American National Standards Committee on Computers and Information Processing, X3. Committee approval of this standard does not necessarily imply that all committee members voted for its approval. At the time it approved this standard, the X3 Committee had the following members:

J. F. Auwaerter, Chairman
R. M. Brown, Vice-Chairman
W. F. Hanrahan, Secretary

Organization	*Name of Representative*
Addressograph Multigraph Corporation	(Representation Vacant)
Air Transport Association	F. C. White
	C. Hart (Alt)
American Library Association	J. R. Rizzolo
	J. C. Kountz (Alt)
	M. S. Malinconico (Alt)
American Nuclear Society	M. L. Couchman
	M. K. Butler (Alt)
	D. R. Vondy (Alt)
Association of American Railroads	R. A. Petrash
Association of Computer Programmers and Analysts	L. A. Ruh
	T. G. Grieb (Alt)
	V. J. Van Dyke (Alt)
Association for Computing Machinery	P. Skelly
	J. A. N. Lee (Alt)
	H. E. Thiess (Alt)
Association of Data Processing Service Organizations	J. B. Christiansen
Association for Educational Data Systems	R. Liquori
Association for Systems Management	W. R. McPherson, Jr
	R. Irwin (Alt)
Association of Time Sharing Users	W. G. Madison
	H. Segal (Alt)
Burroughs Corporation	E. Lohse
	J. S. Foley (Alt)
	J. F. Kalbach (Alt)
California Computer Products, Inc	R. C. Derby
Computer and Communications Industry Association	N. J. Ream
	A. G. W. Biddle (Alt)
Control Data Corporation	C. E. Cooper
	G. I. Williams (Alt)
Data General Corporation	H. Kaikow
	J. Saxena (Alt)

Organization	*Name of Representative*
Datapoint Corporation	H. W. Swanson
	R. J. Stout (Alt)
Data Processing Management Association	A. E. Dubnow
	E. J. Palmer (Alt)
Digital Equipment Computer Users Society	P. Caroom
	B. Ham (Alt)
Digital Equipment Corporation	P. W. White
	A. R. Kent (Alt)
Edison Electric Institute	S. P. Shrivastava
	J. L. Weiser (Alt)
General Services Administration	D. L. Shoemaker
	M. W. Burris (Alt)
GUIDE International	T. E. Wiese
	L. Milligan (Alt)
	D. Stanford (Alt)
Harris Corporation	T. H. Buchert
Honeywell Information Systems, Inc	T. J. McNamara
	E. H. Clamons (Alt)
Institute of Electrical and Electronics Engineers, Communications Society	(Representation Vacant)
Institute of Electrical and Electronics Engineers, Computer Society	T. Feng (Alt)
International Business Machines Corporation	R. J. Holleman
	C. A. Thorn (Alt)
Itel Corporation	R. A. Whitcomb
	R. Baechler (Alt)
Joint Users Group	T. E. Wiese
	R. McQuillian (Alt)
Life Office Management Association	R. E. Ricketts
	J. F. Foley, Jr (Alt)
Litton Industries	I. Danowitz
National Association of State Information Systems	G. I. Theis
	J. L. Lewis (Alt)
National Bureau of Standards	H. S. White, Jr
	R. E. Rountree (Alt)
National Communications System	M. L. Cain
	G. W. White (Alt)
National Machine Tool Builders Association	O. A. Rodriques
NCR Corporation	R. J. Mindlin
	A. R. Daniels (Alt)
	T. W. Kern (Alt)
Olivetti Corporation of America	E. J. Almquist
Printing Industries of America, Inc	N. Scharpf
	E. Rudd (Alt)
Recognition Equipment, Inc	H. F. Schantz
	W. E. Viering (Alt)
Scientific Apparatus Makers Association	A. Savitsky
	J. E. French (Alt)
SHARE Inc	T. B. Steel, Jr
	E. Brubaker (Alt)
	R. H. Wahlen (Alt)
Society of Certified Data Processors	T. M. Kurihara
	A. E. Dubnow (Alt)
Sperry UNIVAC	M. W. Bass
	C. D. Card (Alt)
Telephone Group	V. N. Vaughan, Jr
	S. M. Garland (Alt)
	E. A. Patrick (Alt)
3M Company	R. C. Smith
U.S. Department of Defense	W. L. McGreer
	W. C. Rinehuls (Alt)
	W. B. Robertson (Alt)
U.S. Department of Health, Education, and Welfare	W. R. McPherson, Jr
	W. Frederic (Alt)
VIM	E. Heinze
	M. R. Speers (Alt)
	S. W. White (Alt)
Xerox Corporation	J. L. Wheeler
	A. R. Machell (Alt)

Subcommittee X3J3 on FORTRAN, which developed this standard, had the following members:

F. Engel, Jr, Chairman
M. Greenfield, Vice-Chairman
L. W. Campbell, Secretary
J. C. Noll, International Representative

J. C. Adams
J. T. Bagwell
C. B. Bailey
N. H. Barth
G. M. Bauer
G. A. Beck
G. T. Boswell
W. S. Brainerd
R. F. Brender
J. R. Coleman
J. J. Daly
D. C. Dillon
W. R. Earley
D. R. Eaton
J. T. Engle
D. L. Eriksson
J. C. Flint
M. F. Freeman
C. A. Giammo
T. A. Gibson
R. B. Grove
D. E. Hamilton
J. K. Harkins
D. A. Herington
F. E. Holberton
S. Hue
F. J. Infante
G. W. Johnson
A. R. Jones
R. A. Karp
M. D. Kelley
E. W. Klein
D. T. Laird

R. G. Langsner
J. E. Lauer
S. E. Lovell
N. H. Marshall
B. A. Martin
J. T. Martin
J. H. Matheny
L. P. Meissner
J. J. Mimmack
H. R. Moore
D. J. Olsen
R. L. Page
D. I. Paterson
B. W. Puerling
R. R. Ragan
M. A. Rainer
C. H. Sampson
R. J. Saunders
W. Schenk
R. R. Schieber
J. C. Schwebel
E. H. Senn
K. C. Shih
R. W. Signor
R. T. Slavinski
B. J. Swain
J. F. Thorlin
R. B. Upshaw
D. R. Vondy
J. M. Watson
V. B. Wayland
M. D. Weldon
D. R. Young

Contents

SECTION PAGE

SECTION PAGE

1. INTRODUCTION

1.1 Purpose

This standard specifies the form and establishes the interpretation of programs expressed in the FORTRAN language. The purpose of this standard is to promote portability of FORTRAN programs for use on a variety of data processing systems.

1.2 Processor

The combination of a data processing system and the mechanism by which programs are transformed for use on that data processing system is called a processor in this standard.

1.3 Scope

1.3.1 Inclusions. This standard specifies:

(1) The form of a program written in the FORTRAN language

(2) Rules for interpreting the meaning of such a program and its data

(3) The form of writing input data to be processed by such a program operating on data processing systems

(4) The form of the output data resulting from the use of such a program on data processing systems

1.3.2 Exclusions. This standard does not specify:

(1) The mechanism by which programs are transformed for use on a data processing system

(2) The method of transcription of programs or their input or output data to or from a data processing medium

(3) The operations required for setup and control of the use of programs on data processing systems

(4) The results when the rules of this standard fail to establish an interpretation

(5) The size or complexity of a program and its data that will exceed the capacity of any specific data processing system or the capability of a particular processor

(6) The range or precision of numeric quantities and the method of rounding of numeric results

1. INTRODUCTION

1.1 Purpose

This standard specifies the form and establishes the interpretation of programs expressed in the FORTRAN language. The purpose of this standard is to promote portability of FORTRAN programs for use on a variety of data processing systems.

1.2 Processor

The combination of a data processing system and the mechanism by which programs are transformed for use on that data processing system is called a processor in this standard.

1.3 Scope

1.3.1 Inclusions. This standard specifies:

(1) The form of a program written in the FORTRAN language

(2) Rules for interpreting the meaning of such a program and its data

(3) The form of writing input data to be processed by such a program operating on data processing systems

(4) The form of the output data resulting from the use of such a program on data processing systems

1.3.2 Exclusions. This standard does not specify:

(1) The mechanism by which programs are transformed for use on a data processing system

(2) The method of transcription of programs or their input or output data to or from a data processing medium

(3) The operations required for setup and control of the use of programs on data processing systems

(4) The results when the rules of this standard fail to establish an interpretation

(5) The size or complexity of a program and its data that will exceed the capacity of any specific data processing system or the capability of a particular processor

(6) The range or precision of numeric quantities and the method of rounding of numeric results

(7) The physical properties of input/output records, files, and units

(8) The physical properties and implementation of storage

1.4 Conformance

The requirements, prohibitions, and options specified in this standard generally refer to permissible forms and relationships for standard-conforming programs rather than for processors. The obvious exceptions are the optional output forms produced by a processor, which are not under the control of a program. The requirements, prohibitions, and options for a standard-conforming processor usually must be inferred from those given for programs.

An executable program (2.4.2) conforms to this standard if it uses only those forms and relationships described herein and if the executable program has an interpretation according to this standard. A program unit (2.4) conforms to this standard if it can be included in an executable program in a manner that allows the executable program to be standard conforming.

A processor conforms to this standard if it executes standard-conforming programs in a manner that fulfills the interpretations prescribed herein. A standard-conforming processor may allow additional forms and relationships provided that such additions do not conflict with the standard forms and relationships. However, a standard-conforming processor may allow additional intrinsic functions (15.10) even though this could cause a conflict with the name of an external function in a standard-conforming program. If such a conflict occurs, the processor is permitted to use the intrinsic function unless the name appears in an EXTERNAL statement within the program unit. A standard-conforming program must not use intrinsic functions that have been added by the processor. Note that a standard-conforming program must not use any forms or relationships that are prohibited by this standard, but a standard-conforming processor may allow such forms and relationships if they do not change the proper interpretation of a standard-conforming program.

Because a standard-conforming program may place demands on the processor that are not within the scope of this standard or may include standard items that are not portable, such as external procedures defined by means other than FORTRAN, conformance to this standard does not ensure that a standard-conforming program will execute consistently on all or any standard-conforming processors.

1.4.1 Subset Conformance.

This standard describes two levels of the FORTRAN language, referred to as FORTRAN and subset FORTRAN. FORTRAN is the full language. Subset FORTRAN is a subset of the full language.

(7) The physical properties of input/output records, files, and units

(8) The physical properties and implementation of storage

1.4 Conformance

The requirements, prohibitions, and options specified in this standard generally refer to permissible forms and relationships for standard-conforming programs rather than for processors. The obvious exceptions are the optional output forms produced by a processor, which are not under the control of a program. The requirements, prohibitions, and options for a standard-conforming processor usually must be inferred from those given for programs.

An executable program (2.4.2) conforms to this standard if it uses only those forms and relationships described herein and if the executable program has an interpretation according to this standard. A program unit (2.4) conforms to this standard if it can be included in an executable program in a manner that allows the executable program to be standard conforming.

A processor conforms to this standard if it executes standard-conforming programs in a manner that fulfills the interpretations prescribed herein. A standard-conforming processor may allow additional forms and relationships provided that such additions do not conflict with the standard forms and relationships. However, a standard-conforming processor may allow additional intrinsic functions (15.10) even though this could cause a conflict with the name of an external function in a standard-conforming program. If such a conflict occurs, the processor is permitted to use the intrinsic function unless the name appears in an EXTERNAL statement within the program unit. A standard-conforming program must not use intrinsic functions that have been added by the processor. Note that a standard-conforming program must not use any forms or relationships that are prohibited by this standard, but a standard-conforming processor may allow such forms and relationships if they do not change the proper interpretation of a standard-conforming program.

Because a standard-conforming program may place demands on the processor that are not within the scope of this standard or may include standard items that are not portable, such as external procedures defined by means other than FORTRAN, conformance to this standard does not ensure that a standard-conforming program will execute consistently on all or any standard-conforming processors.

1.4.1 Subset Conformance. This standard describes two levels of the FORTRAN language, referred to as FORTRAN and subset FORTRAN. FORTRAN is the full language. Subset FORTRAN is a subset of the full language.

An executable program conforms to the subset level of this standard if it uses only those forms and relationships described herein for that level and if the executable program has an interpretation according to this standard at that level and would have the same interpretation in the full language. A program unit conforms to the subset level of this standard if it can be included in an executable program in a manner that allows the executable program to be standard conforming at that level.

A subset level processor conforms to the subset level of this standard if it executes subset level standard-conforming programs in a manner that fulfills the interpretations prescribed herein for subset FORTRAN. A subset level processor may include an extension that has a form and would have an interpretation at the full level only if the extension has the interpretation provided by the full level. A subset level processor may also include extensions that do not have forms and interpretations in the full language.

1.5 Notation Used in This Standard

In this standard, "must" is to be interpreted as a requirement; conversely, "must not" is to be interpreted as a prohibition.

In describing the form of FORTRAN statements or constructs, the following metalanguage conventions and symbols are used:

(1) Special characters from the FORTRAN character set, uppercase letters, and uppercase words are to be written as shown, except where otherwise noted.

(2) Lowercase letters and lowercase words indicate general entities for which specific entities must be substituted in actual statements. Once a given lowercase letter or word is used in a syntactic specification to represent an entity, all subsequent occurrences of that letter or word represent the same entity until that letter or word is used in a subsequent syntactic specification to represent a different entity.

(3) Brackets, [ ], are used to indicate optional items.

(4) An ellipsis, ... , indicates that the preceding optional items may appear one or more times in succession.

(5) Blanks are used to improve readability, but unless otherwise noted have no significance.

(6) Words or groups of words that have special significance are underlined where their meaning is

An executable program conforms to the subset level of this standard if it uses only those forms and relationships described herein for that level and if the executable program has an interpretation according to this standard at that level and would have the same interpretation in the full language. A program unit conforms to the subset level of this standard if it can be included in an executable program in a manner that allows the executable program to be standard conforming at that level.

A subset level processor conforms to the subset level of this standard if it executes subset level standard-conforming programs in a manner that fulfills the interpretations prescribed herein for subset FORTRAN. A subset level processor may include an extension that has a form and would have an interpretation at the full level only if the extension has the interpretation provided by the full level. A subset level processor may also include extensions that do not have forms and interpretations in the full language.

1.5 Notation Used in This Standard

In this standard, "must" is to be interpreted as a requirement; conversely, "must not" is to be interpreted as a prohibition.

In describing the form of FORTRAN statements or constructs, the following metalanguage conventions and symbols are used:

(1) Special characters from the FORTRAN character set, uppercase letters, and uppercase words are to be written as shown, except where otherwise noted.

(2) Lowercase letters and lowercase words indicate general entities for which specific entities must be substituted in actual statements. Once a given lowercase letter or word is used in a syntactic specification to represent an entity, all subsequent occurrences of that letter or word represent the same entity until that letter or word is used in a subsequent syntactic specification to represent a different entity.

(3) Brackets, [ ], are used to indicate optional items.

(4) An ellipsis, ... , indicates that the preceding optional items may appear one or more times in succession.

(5) Blanks are used to improve readability, but unless otherwise noted have no significance.

(6) Words or groups of words that have special significance are underlined where their meaning is

described. Titles and the metalanguage symbols described in 1.5(2) are also underlined.

An example illustrates the metalanguage. Given a description of the form of a statement as:

```
CALL sub [( [a [, a]...] )]
```

the following forms are allowed:

```
CALL sub
CALL sub ()
CALL sub (a)
CALL sub (a, a)
CALL sub (a, a, a)
etc
```

When an actual statement is written, specific entities are substituted for sub and each a; for example:

```
CALL ABCD (X,1.0)
```

1.6 Subset Text

The section titles in the subset description are identical to the section titles in the full language description.

There are some instances in which a general situation occurs in the full language but only a restricted case applies to the subset. For example, in 3.6, the "nonexecutable statements" that may appear between executable statements may only be FORMAT statements in the subset. In most of these instances, the more general text of the full language description has been retained in the subset description, even though it is to be interpreted as covering only the restricted case.

To help find differences between the full and subset languages, vertical bars have been added in the margins where the text of the full and subset languages differ.

described. Titles and the metalanguage symbols described in 1.5(2) are also underlined.

An example illustrates the metalanguage. Given a description of the form of a statement as:

```
CALL sub [( [a [, a]...] )]
```

the following forms are allowed:

```
CALL sub
CALL sub ()
CALL sub (a)
CALL sub (a, a)
CALL sub (a, a, a)
etc
```

When an actual statement is written, specific entities are substituted for sub and each a; for example:

```
CALL ABCD (X,1.0)
```

1.6 Subset Text

The section titles in the subset description are identical to the section titles in the full language description.

There are some instances in which a general situation occurs in the full language but only a restricted case applies to the subset. For example, in 3.6, the "nonexecutable statements" that may appear between executable statements may only be FORMAT statements in the subset. In most of these instances, the more general text of the full language description has been retained in the subset description, even though it is to be interpreted as covering only the restricted case.

To help find differences between the full and subset languages, vertical bars have been added in the margins where the text of the full and subset languages differ. For example, this sentence does not appear in the subset language text.

2. FORTRAN TERMS AND CONCEPTS

This section introduces basic terminology and concepts, some of which are clarified further in later sections. Many terms and concepts of more specialized meaning are also introduced in later sections. The underlined words are described here and used throughout this standard.

2.1 Sequence

A sequence is a set ordered by a one-to-one correspondence with the numbers 1, 2, through *n*. The number of elements in the sequence is *n*. A sequence may be empty, in which case it contains no elements.

The elements of a nonempty sequence are referred to as the first element, second element, etc. The *n*th element, where *n* is the number of elements in the sequence, is called the last element. An empty sequence has no first or last element.

2.2 Syntactic Items

Letters, digits, and special characters of the FORTRAN character set (3.1) are used to form the syntactic items of the FORTRAN language. The basic syntactic items of the FORTRAN language are constants, symbolic names, statement labels, keywords, operators, and special characters.

The form of a constant is described in Section 4.

A symbolic name takes the form of a sequence of one to six letters or digits, the first of which must be a letter. Classification of symbolic names and restrictions on their use are described in Section 18.

A statement label takes the form of a sequence of one to five digits, one of which must be nonzero, and is used to identify a statement (3.4).

A keyword takes the form of a specified sequence of letters. The keywords that are significant in the FORTRAN language are described in Sections 7 through 16. In many instances, a keyword or a portion of a keyword also meets the requirements for a symbolic name. Whether a particular sequence of characters identifies a keyword or a symbolic name is implied by context. There is no sequence of characters that is reserved in all contexts in FORTRAN.

The set of special characters is described in 3.1.4. A special character may be an operator or part of a constant or have some other special meaning. The interpretation is implied by context.

2. FORTRAN TERMS AND CONCEPTS

This section introduces basic terminology and concepts, some of which are clarified further in later sections. Many terms and concepts of more specialized meaning are also introduced in later sections. The underlined words are described here and used throughout this standard.

2.1 Sequence

A *sequence* is a set ordered by a one-to-one correspondence with the numbers 1, 2, through *n*. The number of elements in the sequence is *n*. A sequence may be empty, in which case it contains no elements.

The elements of a nonempty sequence are referred to as the first element, second element, etc. The *n*th element, where *n* is the number of elements in the sequence, is called the last element. An empty sequence has no first or last element.

2.2 Syntactic Items

Letters, digits, and special characters of the FORTRAN character set (3.1) are used to form the syntactic items of the FORTRAN language. The basic syntactic items of the FORTRAN language are constants, symbolic names, statement labels, keywords, operators, and special characters.

The form of a constant is described in Section 4.

A *symbolic name* takes the form of a sequence of one to six letters or digits, the first of which must be a letter. Classification of symbolic names and restrictions on their use are described in Section 18.

A *statement label* takes the form of a sequence of one to five digits, one of which must be nonzero, and is used to identify a statement (3.4).

A *keyword* takes the form of a specified sequence of letters. The keywords that are significant in the FORTRAN language are described in Sections 7 through 16. In many instances, a keyword or a portion of a keyword also meets the requirements for a symbolic name. Whether a particular sequence of characters identifies a keyword or a symbolic name is implied by context. There is no sequence of characters that is reserved in all contexts in FORTRAN.

The set of special characters is described in 3.1.4. A special character may be an operator or part of a constant or have some other special meaning. The interpretation is implied by context.

2.3 Statements, Comments, and Lines

A FORTRAN statement is a sequence of syntactic items, as described in Sections 7 through 16. Except for assignment and statement function statements, each statement begins with a keyword. In this standard, the keyword or keywords that begin the statement are used to identify that statement. For example, a DATA statement begins with the keyword DATA.

A statement is written in one or more lines, the first of which is called an initial line (3.2.2); succeeding lines, if any, are called continuation lines (3.2.3).

There is also a line called a comment line (3.2.1), which is not part of any statement and is intended to provide documentation.

2.3.1 Classes of Statements. Each statement is classified as executable or nonexecutable (Section 7). Executable statements specify actions. Nonexecutable statements describe the characteristics, arrangement, and initial values of data; contain editing information; specify statement functions; and classify program units.

2.4 Program Units and Procedures

A program unit consists of a sequence of statements and optional comment lines. A program unit is either a main program or a subprogram.

A main program is a program unit that does not have a FUNCTION or SUBROUTINE statement as its first statement; it may have a PROGRAM statement as its first statement.

A subprogram is a program unit that has a FUNCTION or SUBROUTINE statement as its first statement. A subprogram whose first statement is a FUNCTION statement is called a function subprogram. A subprogram whose first statement is a SUBROUTINE statement is called a subroutine subprogram. Function subprograms and subroutine subprograms are called procedure subprograms.

2.4.1 Procedures. Subroutines (15.6), external functions (15.5), statement functions (15.4), and the intrinsic functions (15.3) are called procedures. Subroutines and external functions are called external procedures. External procedures may also be specified by means other than FORTRAN subprograms.

2.3 Statements, Comments, and Lines

A FORTRAN *statement* is a sequence of syntactic items, as described in Sections 7 through 16. Except for assignment and statement function statements, each statement begins with a keyword. In this standard, the keyword or keywords that begin the statement are used to identify that statement. For example, a DATA statement begins with the keyword DATA.

A statement is written in one or more lines, the first of which is called an *initial line* (3.2.2); succeeding lines, if any, are called *continuation lines* (3.2.3).

There is also a line called a *comment line* (3.2.1), which is not part of any statement and is intended to provide documentation.

2.3.1 Classes of Statements. Each statement is classified as executable or nonexecutable (Section 7). Executable statements specify actions. Nonexecutable statements describe the characteristics, arrangement, and initial values of data; contain editing information; specify statement functions; classify program units; and specify entry points within subprograms.

2.4 Program Units and Procedures

A *program unit* consists of a sequence of statements and optional comment lines. A program unit is either a main program or a subprogram.

A *main program* is a program unit that does not have a FUNCTION, SUBROUTINE, or BLOCK DATA statement as its first statement; it may have a PROGRAM statement as its first statement.

A *subprogram* is a program unit that has a FUNCTION, SUBROUTINE, or BLOCK DATA statement as its first statement. A subprogram whose first statement is a FUNCTION statement is called a *function subprogram*. A subprogram whose first statement is a SUBROUTINE statement is called a *subroutine subprogram*. Function subprograms and subroutine subprograms are called *procedure subprograms*. A subprogram whose first statement is a BLOCK DATA statement is called a *block data subprogram*.

2.4.1 Procedures. Subroutines (15.6), external functions (15.5), statement functions (15.4), and the intrinsic functions (15.3) are called *procedures*. Subroutines and external functions are called *external procedures*. Function subprograms and subroutine subprograms may specify one or more external functions and subroutines, respectively (15.7). External procedures may also be specified by means other than FORTRAN subprograms.

2.4.2 Executable Program. An *executable program* is a collection of program units that consists of exactly one main program and any number, including none, of subprograms and external procedures.

2.5 Variable

A *variable* is an entity that has both a name and a type. A variable name is a symbolic name of a datum. Such a datum may be identified, defined (2.11), and referenced (2.12). Note that the usage in this standard of the word "variable" is more restricted than its normal usage, in that it does not include array elements.

The type of a variable is optionally specified by the appearance of the variable name in a type-statement (8.4). If it is not so specified, the type of a variable is implied by the first letter of the variable name to be integer or real (4.1.2), unless the initial letter type implication is changed by the use of an IMPLICIT statement (8.5).

At any given time during the execution of an executable program, a variable is either defined or undefined (2.11).

2.6 Array

An *array* is a nonempty sequence of data that has a name and a type. The name of an array is a symbolic name.

2.6.1 Array Elements. Each of the elements of an array is called an *array element*. An array name qualified by a subscript is an array element name and identifies a particular element of the array (5.3). Such a datum may be identified, defined (2.11), and referenced (2.12). The number of array elements in an array is specified by an *array declarator* (5.1).

An array element has a type. The type of all array elements within an array is the same, and is optionally specified by the appearance of the array name in a type-statement (8.4). If it is not so specified, the type of an array element is implied by the first letter of the array name to be integer or real (4.1.2), unless the initial letter type implication is changed by the use of an IMPLICIT statement (8.5).

At any given time during the execution of an executable program, an array element is either defined or undefined (2.11).

2.7 Substring

A character datum is a nonempty sequence of characters. A *substring* is a contiguous portion of a character datum. Substring names are not included in the subset.

2.4.2 Executable Program. An *executable program* is a collection of program units that consists of exactly one main program and any number, including none, of subprograms and external procedures.

2.5 Variable

A *variable* is an entity that has both a name and a type. A variable name is a symbolic name of a datum. Such a datum may be identified, defined (2.11), and referenced (2.12). Note that the usage in this standard of the word "variable" is more restricted than its normal usage, in that it does not include array elements.

The type of a variable is optionally specified by the appearance of the variable name in a type-statement (8.4). If it is not so specified, the type of a variable is implied by the first letter of the variable name to be integer or real (4.1.2), unless the initial letter type implication is changed by the use of an IMPLICIT statement (8.5).

At any given time during the execution of an executable program, a variable is either defined or undefined (2.11).

2.6 Array

An *array* is a nonempty sequence of data that has a name and a type. The name of an array is a symbolic name.

2.6.1 Array Elements. Each of the elements of an array is called an *array element*. An array name qualified by a subscript is an array element name and identifies a particular element of the array (5.3). Such a datum may be identified, defined (2.11), and referenced (2.12). The number of array elements in an array is specified by an *array declarator* (5.1).

An array element has a type. The type of all array elements within an array is the same, and is optionally specified by the appearance of the array name in a type-statement (8.4). If it is not so specified, the type of an array element is implied by the first letter of the array name to be integer or real (4.1.2), unless the initial letter type implication is changed by the use of an IMPLICIT statement (8.5).

At any given time during the execution of an executable program, an array element is either defined or undefined (2.11).

2.7 Substring

A character datum is a nonempty sequence of characters. A *substring* is a contiguous portion of a character datum. The form of a substring name used to identify, define (2.11), or reference (2.12) a substring is described in 5.7.1.

2.8 Dummy Argument

A dummy argument in a procedure is a symbolic name. A symbolic name dummy argument identifies a variable, array, or procedure that becomes associated (2.14) with an actual argument of each reference (2.12) to the procedure (15.2, 15.4.2, 15.5.2, and 15.6.2).

Each dummy argument name that is classified as a variable, array, or dummy procedure may appear wherever an actual name of the same class (Section 18) and type may appear, except where explicitly prohibited.

2.9 Scope of Symbolic Names and Statement Labels

The scope of a symbolic name (18.1) is an executable program, a program unit, or a statement function statement.

The name of the main program and the names of external functions, subroutines, and common blocks have a scope of an executable program.

The names of variables, arrays, constants, statement functions, intrinsic functions, and dummy procedures have a scope of a program unit.

The names of variables that appear as dummy arguments in a statement function statement have a scope of that statement.

Statement labels have a scope of a program unit.

2.10 List

A list is a nonempty sequence (2.1) of syntactic entities separated by commas. The entities in the list are called list items.

2.11 Definition Status

At any given time during the execution of an executable program, the definition status of each variable or array element is either defined or undefined (Section 17).

At any given time during the execution of an executable program, a substring is either defined or undefined (2.11).

2.8 Dummy Argument

A dummy argument in a procedure is either a symbolic name or an asterisk. A symbolic name dummy argument identifies a variable, array, or procedure that becomes associated (2.14) with an actual argument of each reference (2.12) to the procedure (15.2, 15.4.2, 15.5.2, and 15.6.2). An asterisk dummy argument indicates that the corresponding actual argument is an alternate return specifier (15.6.2.3, 15.8.3, and 15.9.3.5).

Each dummy argument name that is classified as a variable, array, or dummy procedure may appear wherever an actual name of the same class (Section 18) and type may appear, except where explicitly prohibited.

2.9 Scope of Symbolic Names and Statement Labels

The scope of a symbolic name (18.1) is an executable program, a program unit, a statement function statement, or an implied-DO list in a DATA statement.

The name of the main program and the names of block data subprograms, external functions, subroutines, and common blocks have a scope of an executable program.

The names of variables, arrays, constants, statement functions, intrinsic functions, and dummy procedures have a scope of a program unit.

The names of variables that appear as dummy arguments in a statement function statement have a scope of that statement.

The names of variables that appear as the DO-variable of an implied-DO in a DATA statement have a scope of the implied-DO list.

Statement labels have a scope of a program unit.

2.10 List

A list is a nonempty sequence (2.1) of syntactic entities separated by commas. The entities in the list are called list items.

2.11 Definition Status

At any given time during the execution of an executable program, the definition status of each variable, array element, or substring is either defined or undefined (Section 17).

A defined entity has a value. The value of a defined entity does not change until the entity becomes undefined or is redefined with a different value.

If a variable or array element is undefined, it does not have a predictable value.

A previously defined variable or array element may become undefined. Subsequent definition of a defined variable or array element is permitted, except where it is explicitly prohibited.

A character variable or character array element is defined if every substring of length one of the entity is defined.

An entity is initially defined if it is assigned a value in a DATA statement (Section 9). Initially defined entities are in the defined state at the beginning of execution of an executable program. All variables and array elements not initially defined, or associated (2.14) with an initially defined entity, are undefined at the beginning of execution of an executable program.

An entity must be defined at the time a reference to it is executed.

2.12 Reference

A variable or array element reference is the appearance of a variable or array element name, respectively, in a statement in a context requiring the value of that entity to be used during the execution of the executable program. When a reference to an entity is executed, its current value is available. In this standard, the act of defining an entity is not considered a reference to that entity.

A procedure reference is the appearance of a procedure name in a statement in a context that requires the actions specified by the procedure to be executed during the execution of the executable program. When a procedure reference is executed, the procedure must be available.

2.13 Storage

A storage sequence is a sequence of storage units. A storage unit is either a numeric storage unit or a character storage unit.

An integer, real, or logical datum has one numeric storage unit in a storage sequence. A character datum has one

A defined entity has a value. The value of a defined entity does not change until the entity becomes undefined or is redefined with a different value.

If a variable, array element, or substring is undefined, it does not have a predictable value.

A previously defined variable or array element may become undefined. Subsequent definition of a defined variable or array element is permitted, except where it is explicitly prohibited.

A character variable, character array element, or character substring is defined if every substring of length one of the entity is defined. Note that if a string is defined, every substring of the string is defined, and if any substring of the string is undefined, the string is undefined. Defining any substring does not cause any other string or substring to become undefined.

An entity is *initially defined* if it is assigned a value in a DATA statement (Section 9). Initially defined entities are in the defined state at the beginning of execution of an executable program. All variables and array elements not initially defined, or associated (2.14) with an initially defined entity, are undefined at the beginning of execution of an executable program.

An entity must be defined at the time a reference to it is executed.

2.12 Reference

A variable, array element, or substring *reference* is the appearance of a variable, array element, or substring name, respectively, in a statement in a context requiring the value of that entity to be used during the execution of the executable program. When a reference to an entity is executed, its current value is available. In this standard, the act of defining an entity is not considered a reference to that entity.

A procedure reference is the appearance of a procedure name in a statement in a context that requires the actions specified by the procedure to be executed during the execution of the executable program. When a procedure reference is executed, the procedure must be available.

2.13 Storage

A *storage sequence* is a sequence of storage units. A *storage unit* is either a numeric storage unit or a character storage unit.

An integer, real, or logical datum has one *numeric storage unit* in a storage sequence. A double precision or complex

<u>character storage unit</u> in a storage sequence for each character in the datum. This standard does not specify a relationship between a numeric storage unit and a character storage unit.

If a datum requires more than one storage unit in a storage sequence, those storage units are consecutive.

The concept of a storage sequence is used to describe relationships that exist among variables, array elements, arrays, and common blocks. This standard does not specify a relationship between the storage sequence concept and the physical properties or implementation of storage.

2.14 Association

<u>Association</u> of entities exists if the same datum may be identified by different symbolic names in the same program unit, or by the same name or a different name in different program units of the same executable program (17.1).

Entities may become associated by the following:

(1) Common association (8.3.4)

(2) Equivalence association (8.2.2)

(3) Argument association (15.9.3)

datum has two numeric storage units in a storage sequence. A character datum has one <u>character storage unit</u> in a storage sequence for each character in the datum. This standard does not specify a relationship between a numeric storage unit and a character storage unit.

If a datum requires more than one storage unit in a storage sequence, those storage units are consecutive.

The concept of a storage sequence is used to describe relationships that exist among variables, array elements, arrays, substrings, and common blocks. This standard does not specify a relationship between the storage sequence concept and the physical properties or implementation of storage.

2.14 <u>Association</u>

<u>Association</u> of entities exists if the same datum may be identified by different symbolic names in the same program unit, or by the same name or a different name in different program units of the same executable program (17.1).

Entities may become associated by the following:

(1) Common association (8.3.4)

(2) Equivalence association (8.2.2)

(3) Argument association (15.9.3)

(4) Entry association (15.7.3)

3. CHARACTERS, LINES, AND EXECUTION SEQUENCE

3.1 FORTRAN Character Set

The FORTRAN character set consists of twenty-six letters, ten digits, and eleven special characters.

3.1.1 Letters. A letter is one of the twenty-six characters:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

3.1.2 Digits. A digit is one of the ten characters:

0 1 2 3 4 5 6 7 8 9

A string of digits is interpreted in the decimal base number system when a numeric interpretation is appropriate.

3.1.3 Alphanumeric Characters. An alphanumeric character is a letter or a digit.

3.1.4 Special Characters. A special character is one of the eleven characters:

Character	Name of Character
	Blank
=	Equals
+	Plus
-	Minus
*	Asterisk
/	Slash
(	Left Parenthesis
)	Right Parenthesis
,	Comma
.	Decimal Point
'	Apostrophe

3.1.5 Collating Sequence and Graphics. The order in which the letters are listed in 3.1.1 specifies the collating sequence for the letters; A is less than Z. The order in which the digits are listed in 3.1.2 specifies the collating sequence for the digits; 0 is less than 9. The digits and letters must not be intermixed in the collating sequence; all of the digits must precede A or all of the digits must follow Z. The character blank is less than the letter A and less than the digit 0. The order in which the special characters are listed in 3.1.4 does not imply a collating sequence.

3. CHARACTERS, LINES, AND EXECUTION SEQUENCE

3.1 FORTRAN Character Set

The FORTRAN character set consists of twenty-six letters, ten digits, and thirteen special characters.

3.1.1 Letters. A letter is one of the twenty-six characters:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

3.1.2 Digits. A digit is one of the ten characters:

0 1 2 3 4 5 6 7 8 9

A string of digits is interpreted in the decimal base number system when a numeric interpretation is appropriate.

3.1.3 Alphanumeric Characters. An alphanumeric character is a letter or a digit.

3.1.4 Special Characters. A special character is one of the thirteen characters:

Character	Name of Character
	Blank
=	Equals
+	Plus
-	Minus
*	Asterisk
/	Slash
(	Left Parenthesis
)	Right Parenthesis
,	Comma
.	Decimal Point
$	Currency Symbol
'	Apostrophe
:	Colon

3.1.5 Collating Sequence and Graphics. The order in which the letters are listed in 3.1.1 specifies the collating sequence for the letters; A is less than Z. The order in which the digits are listed in 3.1.2 specifies the collating sequence for the digits; 0 is less than 9. The digits and letters must not be intermixed in the collating sequence; all of the digits must precede A or all of the digits must follow Z. The character blank is less than the letter A and less than the digit 0. The order in which the special characters are listed in 3.1.4 does not imply a collating sequence.

The graphics used for the forty-seven characters must be as given in 3.1.1, 3.1.2, and 3.1.4. However, the style of any graphic is not specified.

3.1.6 Blank Character. With the exception of the uses specified (3.2.2, 3.2.3, 3.3, 4.8, 4.8.1, 13.5.1, and 13.5.2), a blank character within a program unit has no meaning and may be used to improve the appearance of the program unit, subject to the restriction on the number of consecutive continuation lines (3.3).

3.2 Lines

A *line* in a program unit is a sequence of 72 characters. All characters must be from the FORTRAN character set, except as described in 3.2.1, 4.8, 12.2.2, and 13.2.1.

The character positions in a line are called *columns* and are numbered consecutively 1, 2, through 72. The number indicates the sequential position of a character in the line, beginning at the left and proceeding to the right. Lines are ordered by the sequence in which they are presented to the processor. Thus, a program unit consists of a totally ordered set of characters.

3.2.1 Comment Line. A *comment line* is any line that contains a C or an asterisk in column 1, or contains only blank characters in columns 1 through 72. A comment line that contains a C or an asterisk in column 1 may contain any character capable of representation in the processor in columns 2 through 72.

A comment line does not affect the executable program in any way and may be used to provide documentation.

A comment line must be followed immediately by an initial line or another comment line. A comment line must not be followed by a continuation line. Comment lines may precede the initial line of the first statement of any program unit.

3.2.2 Initial Line. An *initial line* is any line that is not a comment line and contains the character blank or the digit 0 in column 6. Columns 1 through 5 may contain a statement label (3.4), or each of the columns 1 through 5 must contain the character blank.

3.2.3 Continuation Line. A *continuation line* is any line that contains any character of the FORTRAN character set other than the character blank or the digit 0 in column 6 and contains only blank characters in columns 1 through 5. A statement must not have more than nine continuation lines.

Except for the currency symbol, the graphics used for the forty-nine characters must be as given in 3.1.1, 3.1.2, and 3.1.4. However, the style of any graphic is not specified.

3.1.6 Blank Character. With the exception of the uses specified (3.2.2, 3.2.3, 3.3, 4.8, 4.8.1, 13.5.1, and 13.5.2), a blank character within a program unit has no meaning and may be used to improve the appearance of the program unit, subject to the restriction on the number of consecutive continuation lines (3.3).

3.2 Lines

A line in a program unit is a sequence of 72 characters. All characters must be from the FORTRAN character set, except as described in 3.2.1, 4.8, 12.2.2, and 13.2.1.

The character positions in a line are called columns and are numbered consecutively 1, 2, through 72. The number indicates the sequential position of a character in the line, beginning at the left and proceeding to the right. Lines are ordered by the sequence in which they are presented to the processor. Thus, a program unit consists of a totally ordered set of characters.

3.2.1 Comment Line. A comment line is any line that contains a C or an asterisk in column 1, or contains only blank characters in columns 1 through 72. A comment line that contains a C or an asterisk in column 1 may contain any character capable of representation in the processor in columns 2 through 72.

A comment line does not affect the executable program in any way and may be used to provide documentation.

Comment lines may appear anywhere in the program unit. Comment lines may precede the initial line of the first statement of any program unit. Comment lines may appear between an initial line and its first continuation line or between two continuation lines.

3.2.2 Initial Line. An initial line is any line that is not a comment line and contains the character blank or the digit 0 in column 6. Columns 1 through 5 may contain a statement label (3.4), or each of the columns 1 through 5 must contain the character blank.

3.2.3 Continuation Line. A continuation line is any line that contains any character of the FORTRAN character set other than the character blank or the digit 0 in column 6 and contains only blank characters in columns 1 through 5. A statement must not have more than nineteen continuation lines.

3.3 Statements

The statements of the FORTRAN language are described in Sections 7 through 16 and are used to form program units. Each statement is written in columns 7 through 72 of an initial line and as many as nine continuation lines. An END statement is written only in columns 7 through 72 of an initial line. No other statement in a program unit may have an initial line that appears to be an END statement. Note that a statement must contain no more than 660 characters. Except as part of a logical IF statement (11.5), no statement may begin on a line that contains any part of the previous statement.

Blank characters preceding, within, or following a statement do not change the interpretation of the statement, except when they appear within the datum strings of character constants or the H or apostrophe edit descriptors in FORMAT statements. However, blank characters do count as characters in the limit of total characters allowed in any one statement.

3.4 Statement Labels

Statement labels provide a means of referring to individual statements. Any statement may be labeled, but only labeled executable statements and FORMAT statements may be referred to by the use of statement labels. The form of a statement label is a sequence of one to five digits, one of which must be nonzero. The statement label may be placed anywhere in columns 1 through 5 of the initial line of the statement. The same statement label must not be given to more than one statement in a program unit. Blanks and leading zeros are not significant in distinguishing between statement labels.

3.5 Order of Statements and Lines

A PROGRAM statement may appear only as the first statement of a main program. The first statement of a subprogram must be either a FUNCTION or SUBROUTINE statement.

Within a program unit that permits the statements:

(1) FORMAT statements may appear anywhere;

(2) all specification statements must precede all DATA statements, statement function statements, and executable statements;

(3) all statement function statements must precede all executable statements; and

(4) all DATA statements must appear after the specification statements and precede all statement function statements and executable statements.

3.3 Statements

The statements of the FORTRAN language are described in Sections 7 through 16 and are used to form program units. Each statement is written in columns 7 through 72 of an initial line and as many as nineteen continuation lines. An END statement is written only in columns 7 through 72 of an initial line. No other statement in a program unit may have an initial line that appears to be an END statement. Note that a statement must contain no more than 1320 characters. Except as part of a logical IF statement (11.5), no statement may begin on a line that contains any part of the previous statement.

Blank characters preceding, within, or following a statement do not change the interpretation of the statement, except when they appear within the datum strings of character constants or the H or apostrophe edit descriptors in FORMAT statements. However, blank characters do count as characters in the limit of total characters allowed in any one statement.

3.4 Statement Labels

Statement labels provide a means of referring to individual statements. Any statement may be labeled, but only labeled executable statements and FORMAT statements may be referred to by the use of statement labels. The form of a statement label is a sequence of one to five digits, one of which must be nonzero. The statement label may be placed anywhere in columns 1 through 5 of the initial line of the statement. The same statement label must not be given to more than one statement in a program unit. Blanks and leading zeros are not significant in distinguishing between statement labels.

3.5 Order of Statements and Lines

A PROGRAM statement may appear only as the first statement of a main program. The first statement of a subprogram must be either a FUNCTION, SUBROUTINE, or BLOCK DATA statement.

Within a program unit that permits the statements:

(1) FORMAT statements may appear anywhere;

(2) all specification statements must precede all DATA statements, statement function statements, and executable statements;

(3) all statement function statements must precede all executable statements;

(4) DATA statements may appear anywhere after the specification statements; and

Within the specification statements of a program unit, IMPLICIT statements must precede all other specification statements.

The last line of a program unit must be an END statement.

Figure 1

Required Order of Statements and Comment Lines

<table>
<tr><td rowspan="6">Comment Lines</td><td colspan="2">PROGRAM, FUNCTION, or SUBROUTINE Statement</td></tr>
<tr><td rowspan="5">FORMAT Statements</td><td>IMPLICIT Statements</td></tr>
<tr><td>Other Specification Statements</td></tr>
<tr><td>DATA Statements</td></tr>
<tr><td>Statement Function Statements</td></tr>
<tr><td>Executable Statements</td></tr>
<tr><td colspan="3">END Statement</td></tr>
</table>

Figure 1 is a diagram of the required order of statements and comment lines for a program unit. Vertical lines delineate varieties of statements that may be interspersed. For example, FORMAT statements may be interspersed with statement function statements and executable statements. Horizontal lines delineate varieties of statements that must not be interspersed. For example, statement function statements must not be interspersed with executable statements. Note that an END statement is also an executable statement and must appear only as the last statement of a program unit.

(5) ENTRY statements may appear anywhere except between a block IF statement and its corresponding END IF statement, or between a DO statement and the terminal statement of its DO-loop.

Within the specification statements of a program unit, IMPLICIT statements must precede all other specification statements except PARAMETER statements. Any specification statement that specifies the type of a symbolic name of a constant must precede the PARAMETER statement that defines that particular symbolic name of a constant; the PARAMETER statement must precede all other statements containing the symbolic names of constants that are defined in the PARAMETER statement.

The last line of a program unit must be an END statement.

Figure 1

Required Order of Statements and Comment Lines

<table>
<tr><td rowspan="5">Comment Lines</td><td colspan="3">PROGRAM, FUNCTION, SUBROUTINE, or BLOCK DATA Statement</td></tr>
<tr><td rowspan="4">FORMAT and ENTRY Statements</td><td rowspan="2">PARAMETER Statements</td><td>IMPLICIT Statements</td></tr>
<tr><td>Other Specification Statements</td></tr>
<tr><td rowspan="2">DATA Statements</td><td>Statement Function Statements</td></tr>
<tr><td>Executable Statements</td></tr>
<tr><td colspan="4">END Statement</td></tr>
</table>

Figure 1 is a diagram of the required order of statements and comment lines for a program unit. Vertical lines delineate varieties of statements that may be interspersed. For example, FORMAT statements may be interspersed with statement function statements and executable statements. Horizontal lines delineate varieties of statements that must not be interspersed. For example, statement function statements must not be interspersed with executable statements. Note that an END statement is also an executable statement and must appear only as the last statement of a program unit.

3.6 Normal Execution Sequence and Transfer of Control

<u>Normal execution sequence</u> is the execution of executable statements in the order in which they appear in a program unit. Execution of an executable program begins with the execution of the first executable statement of the main program. When an external procedure specified in a subprogram is referenced, execution begins with the first executable statement that follows the FUNCTION or SUBROUTINE statement that specifies the referenced procedure as the name of a procedure.

A <u>transfer of control</u> is an alteration of the normal execution sequence. Statements that may cause a transfer of control are:

(1) GO TO

(2) Arithmetic IF

(3) RETURN

(4) STOP

(5) An input/output statement containing an end-of-file specifier

(6) A logical IF statement containing any of the above forms

(7) Block IF and ELSE IF

(8) The last statement, if any, of an IF-block or ELSE IF-block

(9) DO

(10) The terminal statement of a DO-loop

(11) END

The effect of these statements on the execution sequence is described in Sections 11, 12, and 15.

The normal execution sequence is not affected by the appearance of nonexecutable statements or comment lines between executable statements. Execution of a function reference or a CALL statement is not considered a transfer of control in the program unit that contains the reference. Execution of a RETURN or END statement in a referenced procedure, or execution of a transfer of control within a referenced procedure, is not considered a transfer of control in the program unit that contains the reference.

3.6 Normal Execution Sequence and Transfer of Control

Normal execution sequence is the execution of executable statements in the order in which they appear in a program unit. Execution of an executable program begins with the execution of the first executable statement of the main program. When an external procedure specified in a subprogram is referenced, execution begins with the first executable statement that follows the FUNCTION, SUBROUTINE, or ENTRY statement that specifies the referenced procedure as the name of a procedure.

A transfer of control is an alteration of the normal execution sequence. Statements that may cause a transfer of control are:

(1) GO TO

(2) Arithmetic IF

(3) RETURN

(4) STOP

(5) An input/output statement containing an error specifier or end-of-file specifier

(6) CALL with an alternate return specifier

(7) A logical IF statement containing any of the above forms

(8) Block IF and ELSE IF

(9) The last statement, if any, of an IF-block or ELSE IF-block

(10) DO

(11) The terminal statement of a DO-loop

(12) END

The effect of these statements on the execution sequence is described in Sections 11, 12, and 15.

The normal execution sequence is not affected by the appearance of nonexecutable statements or comment lines between executable statements. Execution of a function reference or a CALL statement is not considered a transfer of control in the program unit that contains the reference, except when control is returned to a statement identified by an alternate return specifier in a CALL statement. Execution of a RETURN or END statement in a referenced procedure, or execution of a transfer of control within a

In the execution of an executable program, a procedure subprogram must not be referenced a second time without the prior execution of a RETURN or END statement in that procedure.

referenced procedure, is not considered a transfer of control in the program unit that contains the reference.

In the execution of an executable program, a procedure subprogram must not be referenced a second time without the prior execution of a RETURN or END statement in that procedure.

4. DATA TYPES AND CONSTANTS

4.1 Data Types

The four types of data are:

(1) Integer

(2) Real

(3) Logical

(4) Character

Each type is different and may have a different internal representation. The type may affect the interpretation of the operations involving the datum.

4.1.1 Data Type of a Name. The name employed to identify a datum or a function also identifies its data type. A symbolic name representing a variable, array, or function must have only one type for each program unit. Once a particular name is identified with a particular type in a program unit, that type is implied for any usage of the name in the program unit that requires a type.

4.1.2 Type Rules for Data and Procedure Identifiers. A symbolic name that identifies a variable, array, external function, or statement function may have its type specified in a type-statement (8.4) as integer, real, logical, or character, except that a function may not be of type character. In the absence of an explicit declaration in a type-statement, the type is implied by the first letter of the name. A first letter of I, J, K, L, M, or N implies type integer and any other letter implies type real, unless an IMPLICIT statement (8.5) is used to change the default implied type.

The data type of an array element name is the same as the type of its array name.

The data type of a function name specifies the type of the datum supplied by the function reference in an expression.

A symbolic name that identifies a specific intrinsic function in a program unit has a type as specified in 15.10. An explicit type-statement is not required; however, it is permitted.

4. DATA TYPES AND CONSTANTS

4.1 Data Types

The six types of data are:

(1) Integer

(2) Real

(3) Double precision

(4) Complex

(5) Logical

(6) Character

Each type is different and may have a different internal representation. The type may affect the interpretation of the operations involving the datum.

4.1.1 Data Type of a Name. The name employed to identify a datum or a function also identifies its data type. A symbolic name representing a constant, variable, array, or function (except a generic function) must have only one type for each program unit. Once a particular name is identified with a particular type in a program unit, that type is implied for any usage of the name in the program unit that requires a type.

4.1.2 Type Rules for Data and Procedure Identifiers. A symbolic name that identifies a constant, variable, array, external function, or statement function may have its type specified in a type-statement (8.4) as integer, real, double precision, complex, logical, or character. In the absence of an explicit declaration in a type-statement, the type is implied by the first letter of the name. A first letter of I, J, K, L, M, or N implies type integer and any other letter implies type real, unless an IMPLICIT statement (8.5) is used to change the default implied type.

The data type of an array element name is the same as the type of its array name.

The data type of a function name specifies the type of the datum supplied by the function reference in an expression.

A symbolic name that identifies a specific intrinsic function in a program unit has a type as specified in 15.10. An explicit type-statement is not required; however, it is permitted. A generic function name does not have a predetermined type; the result of a generic function reference assumes a type that depends on the type of the argument, as specified in 15.10. If a generic function name

In a program unit that contains an external function reference, the type of the function is determined in the same manner as for variables and arrays.

The type of an external function is specified implicitly by its name, explicitly in a FUNCTION statement, or explicitly in a type-statement. Note that an IMPLICIT statement within a function subprogram may affect the type of the external function specified in the subprogram.

A symbolic name that identifies a main program, subroutine, or common block has no data type.

4.1.3 Data Type Properties. The mathematical and representation properties for each of the data types are specified in the following sections. For real and integer data, the value zero is considered neither positive nor negative. The value of a signed zero is the same as the value of an unsigned zero.

4.2 Constants

A *constant* is an arithmetic constant, logical constant, or character constant. The value of a constant does not change. Within an executable program, all constants that have the same form have the same value.

4.2.1 Data Type of a Constant. The form of the string representing a constant specifies both its value and data type.

4.2.2 Blanks in Constants. Blank characters occurring in a constant, except in a character constant, have no effect on the value of the constant.

4.2.3 Arithmetic Constants. Integer and real constants are *arithmetic constants*.

4.2.3.1 Signs of Constants. An *unsigned constant* is a constant without a leading sign. A *signed constant* is a constant with a leading plus or minus sign. An *optionally signed constant* is a constant that may be either signed or unsigned. Integer and real constants may be optionally signed constants, except where specified otherwise.

appears in a type-statement, such an appearance is not sufficient by itself to remove the generic properties from that function.

In a program unit that contains an external function reference, the type of the function is determined in the same manner as for variables and arrays.

The type of an external function is specified implicitly by its name, explicitly in a FUNCTION statement, or explicitly in a type-statement. Note that an IMPLICIT statement within a function subprogram may affect the type of the external function specified in the subprogram.

A symbolic name that identifies a main program, subroutine, common block, or block data subprogram has no data type.

4.1.3 Data Type Properties. The mathematical and representation properties for each of the data types are specified in the following sections. For real, double precision, and integer data, the value zero is considered neither positive nor negative. The value of a signed zero is the same as the value of an unsigned zero.

4.2 Constants

A constant is an arithmetic constant, logical constant, or character constant. The value of a constant does not change. Within an executable program, all constants that have the same form have the same value.

4.2.1 Data Type of a Constant. The form of the string representing a constant specifies both its value and data type. A PARAMETER statement (8.6) allows a constant to be given a symbolic name. The symbolic name of a constant must not be used to form part of another constant.

4.2.2 Blanks in Constants. Blank characters occurring in a constant, except in a character constant, have no effect on the value of the constant.

4.2.3 Arithmetic Constants. Integer, real, double precision, and complex constants are arithmetic constants.

4.2.3.1 Signs of Constants. An unsigned constant is a constant without a leading sign. A signed constant is a constant with a leading plus or minus sign. An optionally signed constant is a constant that may be either signed or unsigned. Integer, real, and double precision constants may be optionally signed constants, except where specified otherwise.

4.3 Integer Type

An integer datum is always an exact representation of an integer value. It may assume a positive, negative, or zero value. It may assume only an integral value. An integer datum has one numeric storage unit in a storage sequence.

4.3.1 Integer Constant. The form of an integer constant is an optional sign followed by a nonempty string of digits. The digit string is interpreted as a decimal number.

4.4 Real Type

A real datum is a processor approximation to the value of a real number. It may assume a positive, negative, or zero value. A real datum has one numeric storage unit in a storage sequence.

4.4.1 Basic Real Constant. The form of a basic real constant is an optional sign, an integer part, a decimal point, and a fractional part, in that order. Both the integer part and the fractional part are strings of digits; either of these parts may be omitted but not both. A basic real constant may be written with more digits than a processor will use to approximate the value of the constant. A basic real constant is interpreted as a decimal number.

4.4.2 Real Exponent. The form of a real exponent is the letter E followed by an optionally signed integer constant. A real exponent denotes a power of ten.

4.4.3 Real Constant. The forms of a real constant are:

(1) Basic real constant

(2) Basic real constant followed by a real exponent

(3) Integer constant followed by a real exponent

The value of a real constant that contains a real exponent is the product of the constant that precedes the E and the power of ten indicated by the integer following the E. The integer constant part of form (3) may be written with more digits than a processor will use to approximate the value of the constant.

4.5 Double Precision Type

Double precision type is not included in the subset.

4.3 Integer Type

An integer datum is always an exact representation of an integer value. It may assume a positive, negative, or zero value. It may assume only an integral value. An integer datum has one numeric storage unit in a storage sequence.

4.3.1 Integer Constant. The form of an integer constant is an optional sign followed by a nonempty string of digits. The digit string is interpreted as a decimal number.

4.4 Real Type

A real datum is a processor approximation to the value of a real number. It may assume a positive, negative, or zero value. A real datum has one numeric storage unit in a storage sequence.

4.4.1 Basic Real Constant. The form of a basic real constant is an optional sign, an integer part, a decimal point, and a fractional part, in that order. Both the integer part and the fractional part are strings of digits; either of these parts may be omitted but not both. A basic real constant may be written with more digits than a processor will use to approximate the value of the constant. A basic real constant is interpreted as a decimal number.

4.4.2 Real Exponent. The form of a real exponent is the letter E followed by an optionally signed integer constant. A real exponent denotes a power of ten.

4.4.3 Real Constant. The forms of a real constant are:

(1) Basic real constant

(2) Basic real constant followed by a real exponent

(3) Integer constant followed by a real exponent

The value of a real constant that contains a real exponent is the product of the constant that precedes the E and the power of ten indicated by the integer following the E. The integer constant part of form (3) may be written with more digits than a processor will use to approximate the value of the constant.

4.5 Double Precision Type

A double precision datum is a processor approximation to the value of a real number. The precision, although not specified, must be greater than that of type real. A double precision datum may assume a positive, negative, or zero value. A double precision datum has two consecutive numeric storage units in a storage sequence.

4.5.1 <u>Double Precision Exponent</u>. Double precision type is not included in the subset.

4.5.2 <u>Double Precision Constant</u>. Double precision type is not included in the subset.

4.6 <u>Complex Type</u>

Complex type is not included in the subset.

4.6.1 <u>Complex Constant</u>. Complex type is not included in the subset.

4.7 <u>Logical Type</u>

A logical datum may assume only the values true or false. A logical datum has one numeric storage unit in a storage sequence.

4.5.1 Double Precision Exponent. The form of a *double precision exponent* is the letter D followed by an optionally signed integer constant. A double precision exponent denotes a power of ten. Note that the form and interpretation of a double precision exponent are identical to those of a real exponent, except that the letter D is used instead of the letter E.

4.5.2 Double Precision Constant. The forms of a *double precision constant* are:

(1) Basic real constant followed by a double precision exponent

(2) Integer constant followed by a double precision exponent

The value of a double precision constant is the product of the constant that precedes the D and the power of ten indicated by the integer following the D. The integer constant part of form (2) may be written with more digits than a processor will use to approximate the value of the constant.

4.6 Complex Type

A complex datum is a processor approximation to the value of a complex number. The representation of a complex datum is in the form of an ordered pair of real data. The first of the pair represents the real part of the complex datum and the second represents the imaginary part. Each part has the same degree of approximation as for a real datum. A complex datum has two consecutive numeric storage units in a storage sequence; the first storage unit is the real part and the second storage unit is the imaginary part.

4.6.1 Complex Constant. The form of a *complex constant* is a left parenthesis followed by an ordered pair of real or integer constants separated by a comma, and followed by a right parenthesis. The first constant of the pair is the real part of the complex constant and the second is the imaginary part.

4.7 Logical Type

A logical datum may assume only the values true or false. A logical datum has one numeric storage unit in a storage sequence.

4.7.1 Logical Constant. The forms and values of a logical constant are:

Form	Value
.TRUE.	true
.FALSE.	false

4.8 Character Type

A character datum is a string of characters. The string may consist of any characters capable of representation in the processor. The blank character is valid and significant in a character datum. The length of a character datum is the number of characters in the string. A character datum has one character storage unit in a storage sequence for each character in the string.

Each character in the string has a character position that is numbered consecutively 1, 2, 3, etc. The number indicates the sequential position of a character in the string, beginning at the left and proceeding to the right.

4.8.1 Character Constant. The form of a character constant is an apostrophe followed by a nonempty string of characters followed by an apostrophe. The string may consist of any characters capable of representation in the processor. Note that the delimiting apostrophes are not part of the datum represented by the constant. An apostrophe within the datum string is represented by two consecutive apostrophes with no intervening blanks. In a character constant, blanks embedded between the delimiting apostrophes are significant.

The length of a character constant is the number of characters between the delimiting apostrophes, except that each pair of consecutive apostrophes counts as a single character. The delimiting apostrophes are not counted. The length of a character constant must be greater than zero.

4.7.1 Logical Constant. The forms and values of a logical constant are:

Form	Value
.TRUE.	true
.FALSE.	false

4.8 Character Type

A character datum is a string of characters. The string may consist of any characters capable of representation in the processor. The blank character is valid and significant in a character datum. The length of a character datum is the number of characters in the string. A character datum has one character storage unit in a storage sequence for each character in the string.

Each character in the string has a character position that is numbered consecutively 1, 2, 3, etc. The number indicates the sequential position of a character in the string, beginning at the left and proceeding to the right.

4.8.1 Character Constant. The form of a character constant is an apostrophe followed by a nonempty string of characters followed by an apostrophe. The string may consist of any characters capable of representation in the processor. Note that the delimiting apostrophes are not part of the datum represented by the constant. An apostrophe within the datum string is represented by two consecutive apostrophes with no intervening blanks. In a character constant, blanks embedded between the delimiting apostrophes are significant.

The length of a character constant is the number of characters between the delimiting apostrophes, except that each pair of consecutive apostrophes counts as a single character. The delimiting apostrophes are not counted. The length of a character constant must be greater than zero.

5. ARRAYS AND SUBSTRINGS

An array is a nonempty sequence of data. An array element is one member of the sequence of data. An array name is the symbolic name of an array. An array element name is an array name qualified by a subscript (5.3).

An array name not qualified by a subscript identifies the entire sequence of elements of the array in certain forms where such use is permitted (5.6); however, in an EQUIVALENCE statement, an array name not qualified by a subscript identifies the first element of the array (8.2.4).

An array element name identifies one element of the sequence. The subscript value (Table 1) specifies the element of the array being identified. A different array element may be identified by changing the subscript value of the array element name.

An array name is local to a program unit (18.1.2).

Substrings are not included in the subset.

5.1 Array Declarator

An array declarator specifies a symbolic name that identifies an array within a program unit and specifies certain properties of the array. Only one array declarator for an array name is permitted in a program unit.

5.1.1 Form of an Array Declarator. The form of an array declarator is:

a (d [,d]...)

where: a is the symbolic name of the array

d is a dimension declarator

The number of dimensions of the array is the number of dimension declarators in the array declarator. The minimum number of dimensions is one and the maximum is three.

5.1.1.1 Form of a Dimension Declarator. The form of a dimension declarator is:

d

where d is an integer constant or an integer variable name, called the upper dimension bound. The lower dimension bound is one. The upper dimension bound of the last dimension may be an asterisk in assumed-size array declarators (5.1.2). Integer variables may appear in dimension bounds only in adjustable array declarators (5.1.2).

5. ARRAYS AND SUBSTRINGS

An *array* is a nonempty sequence of data. An *array element* is one member of the sequence of data. An *array name* is the symbolic name of an array. An *array element name* is an array name qualified by a subscript (5.3).

An array name not qualified by a subscript identifies the entire sequence of elements of the array in certain forms where such use is permitted (5.6); however, in an EQUIVALENCE statement, an array name not qualified by a subscript identifies the first element of the array (8.2.4).

An array element name identifies one element of the sequence. The subscript value (Table 1) specifies the element of the array being identified. A different array element may be identified by changing the subscript value of the array element name.

An array name is local to a program unit (18.1.2).

A *substring* is a contiguous portion of a character datum.

5.1 Array Declarator

An *array declarator* specifies a symbolic name that identifies an array within a program unit and specifies certain properties of the array. Only one array declarator for an array name is permitted in a program unit.

5.1.1 Form of an Array Declarator.

The form of an array declarator is:

a (*d* [,*d*]...)

where: *a* is the symbolic name of the array

d is a dimension declarator

The number of dimensions of the array is the number of dimension declarators in the array declarator. The minimum number of dimensions is one and the maximum is seven.

5.1.1.1 Form of a Dimension Declarator.

The form of a *dimension declarator* is:

[d_1:] d_2

where: d_1 is the lower dimension bound

d_2 is the upper dimension bound

The lower and upper dimension bounds are arithmetic expressions, called *dimension bound expressions*, in which all constants, symbolic names of constants, and variables are of type integer. The upper dimension bound of the last

If a variable that appears in a dimension bound is not of default implied integer type (4.1.2), it must be specified as integer by an IMPLICIT statement or a type-statement prior to its appearance in a dimension bound.

5.1.1.2 <u>Value of Dimension Bounds</u>. The value of the upper dimension bound must be greater than or equal to one. An upper dimension bound of an asterisk is always greater than or equal to one.

5.1.2 <u>Kinds and Occurrences of Array Declarators</u>. Each array declarator is either a constant array declarator, an adjustable array declarator, or an assumed-size array declarator. A <u>constant array declarator</u> is an array declarator in which each of the dimension bounds is an integer constant. An <u>adjustable array declarator</u> is an array declarator that contains one or more variables. An <u>assumed-size array declarator</u> is a constant array declarator or an adjustable array declarator, except that the upper dimension bound of the last dimension is an asterisk.

Each array declarator is either an actual array declarator or a dummy array declarator.

5.1.2.1 <u>Actual Array Declarator</u>. An <u>actual array declarator</u> is an array declarator in which the array name is not a dummy argument. Each actual array declarator must be a constant array declarator. An actual array declarator is permitted in a DIMENSION statement, type-statement, or COMMON statement (Section 8).

5.1.2.2 <u>Dummy Array Declarator</u>. A <u>dummy array declarator</u> is an array declarator in which the array name is a dummy argument. A dummy array declarator may be either a constant array declarator, an adjustable array declarator, or an assumed-size array declarator. A dummy array declarator is permitted in a DIMENSION statement or a type-statement but not in a COMMON statement. A dummy array declarator may appear only in a function or subroutine subprogram.

5.2 <u>Properties of an Array</u>

The following properties of an array are specified by the array declarator: the number of dimensions of the array, the dimension may be an asterisk in assumed-size array declarators (5.1.2). A dimension bound expression must not contain a function or array element reference. Integer variables may appear in dimension bound expressions only in adjustable array declarators (5.1.2).

If the symbolic name of a constant or variable that appears in a dimension bound expression is not of default implied integer type (4.1.2), it must be specified as integer by an IMPLICIT statement or a type-statement prior to its appearance in a dimension bound expression.

5.1.1.2 Value of Dimension Bounds. The value of either dimension bound may be positive, negative, or zero; however, the value of the upper dimension bound must be greater than or equal to the value of the lower dimension bound. If only the upper dimension bound is specified, the value of the lower dimension bound is one. An upper dimension bound of an asterisk is always greater than or equal to the lower dimension bound.

5.1.2 Kinds and Occurrences of Array Declarators. Each array declarator is either a constant array declarator, an adjustable array declarator, or an assumed-size array declarator. A constant array declarator is an array declarator in which each of the dimension bound expressions is an integer constant expression (6.1.3.1). An adjustable array declarator is an array declarator that contains one or more variables. An assumed-size array declarator is a constant array declarator or an adjustable array declarator, except that the upper dimension bound of the last dimension is an asterisk.

Each array declarator is either an actual array declarator or a dummy array declarator.

5.1.2.1 Actual Array Declarator. An actual array declarator is an array declarator in which the array name is not a dummy argument. Each actual array declarator must be a constant array declarator. An actual array declarator is permitted in a DIMENSION statement, type-statement, or COMMON statement (Section 8).

5.1.2.2 Dummy Array Declarator. A dummy array declarator is an array declarator in which the array name is a dummy argument. A dummy array declarator may be either a constant array declarator, an adjustable array declarator, or an assumed-size array declarator. A dummy array declarator is permitted in a DIMENSION statement or a type-statement but not in a COMMON statement. A dummy array declarator may appear only in a function or subroutine subprogram.

5.2 Properties of an Array

The following properties of an array are specified by the array declarator: the number of dimensions of the array, the size and bounds of each dimension, and therefore the number of array elements.

The properties of an array in a program unit are specified by the array declarator for the array in that program unit.

5.2.1 Data Type of an Array and an Array Element. An array name has a data type (4.1.1). An array element name has the same data type as the array name.

5.2.2 Dimensions of an Array. The number of dimensions of an array is equal to the number of dimension declarators in the array declarator.

The size of a dimension is the value of $\underline{d}$ where $\underline{d}$ is the value of the upper dimension bound.

The size of a dimension whose upper bound is an asterisk is not specified.

The number and size of dimensions in one array declarator may be different from the number and size of dimensions in another array declarator that is associated by common, equivalence, or argument association.

5.2.3 Size of an Array. The size of an array is equal to the number of elements in the array. The size of an array is equal to the product of the sizes of the dimensions specified by the array declarator for that array name. The size of an assumed-size dummy array (5.5) is determined as follows:

(1) If the actual argument corresponding to the dummy array is a noncharacter or character array name, the size of the dummy array is the size of the actual argument array.

(2) If the actual argument corresponding to the dummy array name is a noncharacter or character array element name with a subscript value of $\underline{r}$ in an array of size $\underline{x}$, the size of the dummy array is $\underline{x} + 1 - \underline{r}$.

size and bounds of each dimension, and therefore the number of array elements.

The properties of an array in a program unit are specified by the array declarator for the array in that program unit.

5.2.1 Data Type of an Array and an Array Element. An array name has a data type (4.1.1). An array element name has the same data type as the array name.

5.2.2 Dimensions of an Array. The number of dimensions of an array is equal to the number of dimension declarators in the array declarator.

The size of a dimension is the value:

$$d_2 - d_1 + 1$$

where: d_1 is the value of the lower dimension bound

d_2 is the value of the upper dimension bound

Note that if the value of the lower dimension bound is one, the size of the dimension is d_2.

The size of a dimension whose upper bound is an asterisk is not specified.

The number and size of dimensions in one array declarator may be different from the number and size of dimensions in another array declarator that is associated by common, equivalence, or argument association.

5.2.3 Size of an Array. The size of an array is equal to the number of elements in the array. The size of an array is equal to the product of the sizes of the dimensions specified by the array declarator for that array name. The size of an assumed-size dummy array (5.5) is determined as follows:

(1) If the actual argument corresponding to the dummy array is a noncharacter array name, the size of the dummy array is the size of the actual argument array.

(2) If the actual argument corresponding to the dummy array name is a noncharacter array element name with a subscript value of r in an array of size x, the size of the dummy array is $x + 1 - r$.

(3) If the actual argument is a character array name, character array element name, or character array element substring name and begins at character storage unit t of an array with c character storage units, then the size of the dummy array is

If an assumed-size dummy array has _n_ dimensions, the product of the sizes of the first _n_ - 1 dimensions must be less than or equal to the size of the array, as determined by one of the immediately preceding rules.

5.2.4 Array Element Ordering. The elements of an array are ordered in a sequence (2.1). An array element name contains a subscript (5.4.1) whose subscript value (5.4.3) determines which element of the array is identified by the array element name. The first element of the array has a subscript value of one; the second element has a subscript value of two; the last element has a subscript value equal to the size of the array.

Whenever an array name unqualified by a subscript is used to designate the whole array (5.6), the appearance of the array name implies that the number of values to be processed is equal to the number of elements in the array and that the elements of the array are to be taken in sequential order.

5.2.5 Array Storage Sequence. An array has a storage sequence consisting of the storage sequences of the array elements in the order determined by the array element ordering. The number of storage units in an array is _x_*_z_, where _x_ is the number of the elements in the array and _z_ is the number of storage units for each array element.

5.3 Array Element Name

The form of an array element name is:

_a_ (_s_ [,_s_]...)

where: _a_ is the array name

(_s_ [,_s_]...) is a subscript (5.4.1)

_s_ is a subscript expression (5.4.2)

The number of subscript expressions must be equal to the number of dimensions in the array declarator for the array name.

5.4 Subscript

5.4.1 Form of a Subscript. The form of a subscript is:

(_s_ [,_s_]...)

where _s_ is a subscript expression.

Note that the term "subscript" includes the parentheses that delimit the list of subscript expressions.

INT((*c* + 1 - *t*) / *ln*), where *ln* is the length of an element of the dummy array.

If an assumed-size dummy array has *n* dimensions, the product of the sizes of the first *n* - 1 dimensions must be less than or equal to the size of the array, as determined by one of the immediately preceding rules.

5.2.4 Array Element Ordering. The elements of an array are ordered in a sequence (2.1). An array element name contains a subscript (5.4.1) whose subscript value (5.4.3) determines which element of the array is identified by the array element name. The first element of the array has a subscript value of one; the second element has a subscript value of two; the last element has a subscript value equal to the size of the array.

Whenever an array name unqualified by a subscript is used to designate the whole array (5.6), the appearance of the array name implies that the number of values to be processed is equal to the number of elements in the array and that the elements of the array are to be taken in sequential order.

5.2.5 Array Storage Sequence. An array has a storage sequence consisting of the storage sequences of the array elements in the order determined by the array element ordering. The number of storage units in an array is *x***z*, where *x* is the number of the elements in the array and *z* is the number of storage units for each array element.

5.3 Array Element Name

The form of an array element name is:

a (*s* [,*s*]...)

where: *a* is the array name

(*s* [,*s*]...) is a subscript (5.4.1)

s is a subscript expression (5.4.2)

The number of subscript expressions must be equal to the number of dimensions in the array declarator for the array name.

5.4 Subscript

5.4.1 Form of a Subscript. The form of a subscript is:

(*s* [,*s*]...)

where *s* is a subscript expression.

Note that the term "subscript" includes the parentheses that delimit the list of subscript expressions.

5.4.2 Subscript Expression. A subscript expression is an integer expression. A subscript expression must not contain array element references and function references.

Within a program unit, the value of each subscript expression must be greater than or equal to one. The value of each subscript expression must not exceed the corresponding upper dimension bound declared for the array in the program unit. If the upper dimension bound is an asterisk, the value of the corresponding subscript expression must be such that the subscript value does not exceed the size of the dummy array.

5.4.3 Subscript Value. The subscript value of a subscript is specified in Table 1. The subscript value determines which array element is identified by the array element name. Within a program unit, the subscript value depends on the values of the subscript expressions in the subscript and on the dimensions of the array specified in the array declarator for the array in the program unit. If the subscript value is *r*, the *r*th element of the array is identified.

5.4.2 Subscript Expression. A *subscript expression* is an integer expression. A subscript expression may contain array element references and function references. Note that a restriction in the evaluation of expressions (6.6) prohibits certain side effects. In particular, evaluation of a function must not alter the value of any other subscript expression within the same subscript.

Within a program unit, the value of each subscript expression must be greater than or equal to the corresponding lower dimension bound in the array declarator for the array. The value of each subscript expression must not exceed the corresponding upper dimension bound declared for the array in the program unit. If the upper dimension bound is an asterisk, the value of the corresponding subscript expression must be such that the subscript value does not exceed the size of the dummy array.

5.4.3 Subscript Value. The *subscript value* of a subscript is specified in Table 1. The subscript value determines which array element is identified by the array element name. Within a program unit, the subscript value depends on the values of the subscript expressions in the subscript and on the dimensions of the array specified in the array declarator for the array in the program unit. If the subscript value is *r*, the *r*th element of the array is identified.

Table 1

Subscript Value

n	Dimension Declarator	Subscript	Subscript Value
1	(d_1)	(s_1)	s_1
2	(d_1,d_2)	(s_1,s_2)	$1+(s_1-1)$ $+(s_2-1)*d_1$
3	(d_1,d_2,d_3)	(s_1,s_2,s_3)	$1+(s_1-1)$ $+(s_2-1)*d_1$ $+(s_3-1)*d_1*d_2$

Notes for Table 1:

(1) n is the number of dimensions, $1 \le n \le 3$.

(2) d_i is the value of the upper bound of the ith dimension. d_i is also the size of the ith dimension.

(3) s_i is the integer value of the ith subscript expression.

Table 1

Subscript Value

n	Dimension Declarator	Subscript	Subscript Value
1	$(j_1:k_1)$	(s_1)	$1+(s_1-j_1)$
2	$(j_1:k_1,j_2:k_2)$	(s_1,s_2)	$1+(s_1-j_1)$ $+(s_2-j_2)*d_1$
3	$(j_1:k_1,j_2:k_2,j_3:k_3)$	(s_1,s_2,s_3)	$1+(s_1-j_1)$ $+(s_2-j_2)*d_1$ $+(s_3-j_3)*d_2*d_1$
. . .	. . .	. . .	. . .
n	$(j_1:k_1,...,j_n:k_n)$	$(s_1,...,s_n)$	$1+(s_1-j_1)$ $+(s_2-j_2)*d_1$ $+(s_3-j_3)*d_2*d_1$ $+...$ $+(s_n-j_n)*d_{n-1}$ $*d_{n-2}*...*d_1$

Notes for Table 1:

(1) n is the number of dimensions, $1 \leq n \leq 7$.

(2) j_i is the value of the lower bound of the ith dimension.

(3) k_i is the value of the upper bound of the ith dimension.

(4) If only the upper bound is specified, then $j_i = 1$.

(5) s_i is the integer value of the ith subscript expression.

(6) $d_i = k_i-j_i+1$ is the size of the ith dimension. If the value of the lower bound is 1, then $d_i = k_i$.

Note that a subscript of (1), (1,1), or (1,1,1) has a subscript value of one and identifies the first element of the array. A subscript of the form (d_1,...,d_n) identifies the last element of the array; its subscript value is equal to the number of elements in the array.

The subscript value and the subscript expression value are not necessarily the same. In the example:

```
      DIMENSION A(10),B(10,10)
      A(2) = B(1,2)
```

A(2) identifies the second element of A, the subscript is (2) with a subscript value of two, and the subscript expression is 2 with a value of two. B(1,2) identifies the eleventh element of B, the subscript is (1,2) with a subscript value of eleven, and the subscript expressions are 1 and 2 with values of one and two.

5.5 Dummy and Actual Arrays

A _dummy array_ is an array for which the array declarator is a dummy array declarator. An _assumed-size dummy array_ is a dummy array for which the array declarator is an assumed-size array declarator. A dummy array is permitted only in a function or subroutine subprogram (Section 15).

An _actual array_ is an array for which the array declarator is an actual array declarator. Each array in the main program is an actual array and must have a constant array declarator. A dummy array may be used as an actual argument.

5.5.1 _Adjustable Arrays and Adjustable Dimensions_. An _adjustable array_ is an array for which the array declarator is an adjustable array declarator. In an adjustable array declarator, those dimension declarators that contain a variable name are called _adjustable dimensions_.

An adjustable array declarator must be a dummy array declarator. The array name must appear in the dummy argument list of the subprogram. A variable name that appears in a dimension bound of an array must also appear as a name either in the dummy argument list or in a common block in that subprogram.

At the time of execution of a reference to a function or subroutine containing an adjustable array in its dummy argument list, each actual argument that corresponds to a dummy argument appearing in a dimension bound for the array and each variable in common appearing in a dimension bound for the array must be defined with an integer value. The values of those dummy arguments or variables in common determine the size of the corresponding adjustable dimension

Note that a subscript of the form $(j_1,...,j_n)$ has a subscript value of one and identifies the first element of the array. A subscript of the form $(k_1,...,k_n)$ identifies the last element of the array; its subscript value is equal to the number of elements in the array.

The subscript value and the subscript expression value are not necessarily the same, even for a one-dimensional array. In the example:

```
      DIMENSION A(-1:8),B(10,10)
      A(2) = B(1,2)
```

A(2) identifies the fourth element of A, the subscript is (2) with a subscript value of four, and the subscript expression is 2 with a value of two. B(1,2) identifies the eleventh element of B, the subscript is (1,2) with a subscript value of eleven, and the subscript expressions are 1 and 2 with values of one and two.

5.5 Dummy and Actual Arrays

A *dummy array* is an array for which the array declarator is a dummy array declarator. An *assumed-size dummy array* is a dummy array for which the array declarator is an assumed-size array declarator. A dummy array is permitted only in a function or subroutine subprogram (Section 15).

An *actual array* is an array for which the array declarator is an actual array declarator. Each array in the main program is an actual array and must have a constant array declarator. A dummy array may be used as an actual argument.

5.5.1 Adjustable Arrays and Adjustable Dimensions.

An *adjustable array* is an array for which the array declarator is an adjustable array declarator. In an adjustable array declarator, those dimension declarators that contain a variable name are called *adjustable dimensions*.

An adjustable array declarator must be a dummy array declarator. At least one dummy argument list of the subprogram must contain the name of the adjustable array. A variable name that appears in a dimension bound expression of an array must also appear as a name either in every dummy argument list that contains the array name or in a common block in that subprogram.

At the time of execution of a reference to a function or subroutine containing an adjustable array in its dummy argument list, each actual argument that corresponds to a dummy argument appearing in a dimension bound expression for the array and each variable in common appearing in a dimension bound expression for the array must be defined with an integer value. The values of those dummy arguments or variables in common, together with any constants and

5.7 Character Substring

Substrings are not included in the subset.

5.7.1 Substring Name. Substrings are not included in the subset.

5.7.2 Substring Expression. Substrings are not included in the subset.

(11) In a SAVE statement

5.7 Character Substring

A character substring is a contiguous portion of a character datum and is of type character. A character substring is identified by a substring name and may be assigned values and referenced.

5.7.1 Substring Name.

The forms of a *substring name* are:

v ([e_1] : [e_2])

a (s [,s]...)([e_1] : [e_2])

where: v is a character variable name

a (s [,s]...) is a character array element name

e_1 and e_2 are each an integer expression and are called *substring expressions*

The value e_1 specifies the leftmost character position of the substring, and the value e_2 specifies the rightmost character position. For example, A(2:4) specifies characters in positions two through four of the character variable A, and B(4,3)(1:6) specifies characters in positions one through six of the character array element B(4,3).

The values of e_1 and e_2 must be such that:

$$1 \le e_1 \le e_2 \le len$$

where *len* is the length of the character variable or array element (8.4.2). If e_1 is omitted, a value of one is implied for e_1. If e_2 is omitted, a value of *len* is implied for e_2. Both e_1 and e_2 may be omitted; for example, the form v(:) is equivalent to v, and the form a(s [,s]...)(:) is equivalent to a(s [,s]...). The length of a character substring is $e_2 - e_1 + 1$.

5.7.2 Substring Expression.

A *substring expression* may be any integer expression. A substring expression may contain array element references and function references. Note that a restriction in the evaluation of expressions (6.6) prohibits certain side effects. In particular, evaluation of a function must not alter the value of any other expression within the same substring name.

6. EXPRESSIONS

This section describes the formation, interpretation, and evaluation rules for arithmetic, character, relational, and logical expressions. An expression is formed from operands, operators, and parentheses.

6.1 Arithmetic Expressions

An arithmetic expression is used to express a numeric computation. Evaluation of an arithmetic expression produces a numeric value.

The simplest form of an arithmetic expression is an unsigned arithmetic constant, arithmetic variable reference, arithmetic array element reference, or arithmetic function reference. More complicated arithmetic expressions may be formed by using one or more arithmetic operands together with arithmetic operators and parentheses. Arithmetic operands must identify values of type integer or real.

6.1.1 Arithmetic Operators. The five arithmetic operators are:

Operator	Representing
**	Exponentiation
/	Division
*	Multiplication
-	Subtraction or Negation
+	Addition or Identity

Each of the operators **, /, and * operates on a pair of operands and is written between the two operands. Each of the operators + and - either:

(1) operates on a pair of operands and is written between the two operands, or

(2) operates on a single operand and is written preceding that operand.

6.1.2 Form and Interpretation of Arithmetic Expressions. The interpretation of the expression formed with each of the arithmetic operators in each form of use is as follows:

6. EXPRESSIONS

This section describes the formation, interpretation, and evaluation rules for arithmetic, character, relational, and logical expressions. An expression is formed from operands, operators, and parentheses.

6.1 Arithmetic Expressions

An arithmetic expression is used to express a numeric computation. Evaluation of an arithmetic expression produces a numeric value.

The simplest form of an arithmetic expression is an unsigned arithmetic constant, symbolic name of an arithmetic constant, arithmetic variable reference, arithmetic array element reference, or arithmetic function reference. More complicated arithmetic expressions may be formed by using one or more arithmetic operands together with arithmetic operators and parentheses. Arithmetic operands must identify values of type integer, real, double precision, or complex.

6.1.1 Arithmetic Operators. The five arithmetic operators are:

Operator	Representing
**	Exponentiation
/	Division
*	Multiplication
-	Subtraction or Negation
+	Addition or Identity

Each of the operators **, /, and * operates on a pair of operands and is written between the two operands. Each of the operators + and - either:

(1) operates on a pair of operands and is written between the two operands, or

(2) operates on a single operand and is written preceding that operand.

6.1.2 Form and Interpretation of Arithmetic Expressions. The interpretation of the expression formed with each of the arithmetic operators in each form of use is as follows:

Use of Operator	Interpretation
x_1 ** x_2	Exponentiate x_1 to the power x_2
x_1 / x_2	Divide x_1 by x_2
x_1 * x_2	Multiply x_1 and x_2
x_1 - x_2	Subtract x_2 from x_1
- x_2	Negate x_2
x_1 + x_2	Add x_1 and x_2
+ x_2	Same as x_2

where: x_1 denotes the operand to the left of the operator

x_2 denotes the operand to the right of the operator

The interpretation of a division may depend on the data types of the operands (6.1.5).

A set of formation rules is used to establish the interpretation of an arithmetic expression that contains two or more operators. There is a precedence among the arithmetic operators, which determines the order in which the operands are to be combined unless the order is changed by the use of parentheses. The precedence of the arithmetic operators is as follows:

Operator	Precedence
**	Highest
* and /	Intermediate
+ and -	Lowest

For example, in the expression

```
- A ** 2
```

the exponentiation operator (**) has precedence over the negation operator (-); therefore, the operands of the exponentiation operator are combined to form an expression that is used as the operand of the negation operator. The interpretation of the above expression is the same as the interpretation of the expression

```
- (A ** 2)
```

The <u>arithmetic operands</u> are:

Use of Operator	Interpretation
x_1 ** x_2	Exponentiate x_1 to the power x_2
x_1 / x_2	Divide x_1 by x_2
x_1 * x_2	Multiply x_1 and x_2
x_1 - x_2	Subtract x_2 from x_1
- x_2	Negate x_2
x_1 + x_2	Add x_1 and x_2
+ x_2	Same as x_2

where: x_1 denotes the operand to the left of the operator

x_2 denotes the operand to the right of the operator

The interpretation of a division may depend on the data types of the operands (6.1.5).

A set of formation rules is used to establish the interpretation of an arithmetic expression that contains two or more operators. There is a precedence among the arithmetic operators, which determines the order in which the operands are to be combined unless the order is changed by the use of parentheses. The precedence of the arithmetic operators is as follows:

Operator	Precedence
**	Highest
* and /	Intermediate
+ and -	Lowest

For example, in the expression

```
- A ** 2
```

the exponentiation operator (**) has precedence over the negation operator (-); therefore, the operands of the exponentiation operator are combined to form an expression that is used as the operand of the negation operator. The interpretation of the above expression is the same as the interpretation of the expression

```
- (A ** 2)
```

The <u>arithmetic operands</u> are:

(1) Primary

(2) Factor

(3) Term

(4) Arithmetic expression

The formation rules to be applied in establishing the interpretation of arithmetic expressions are in 6.1.2.1 through 6.1.2.4.

6.1.2.1 <u>Primaries</u>. The <u>primaries</u> are:

(1) Unsigned arithmetic constant (4.2.3)

(2) Arithmetic variable reference (2.5)

(3) Arithmetic array element reference (5.3)

(4) Arithmetic function reference (15.2)

(5) Arithmetic expression enclosed in parentheses (6.1.2.4)

6.1.2.2 <u>Factor</u>. The forms of a <u>factor</u> are:

(1) Primary

(2) Primary ** factor

Thus, a factor is formed from a sequence of one or more primaries separated by the exponentiation operator. Form (2) indicates that in interpreting a factor containing two or more exponentiation operators, the primaries are combined from right to left. For example, the factor

```
2**3**2
```

has the same interpretation as the factor

```
2**(3**2)
```

6.1.2.3 <u>Term</u>. The forms of a <u>term</u> are:

(1) Factor

(2) Term / factor

(3) Term * factor

Thus, a term is formed from a sequence of one or more factors separated by either the multiplication operator or the division operator. Forms (2) and (3) indicate that in

(1) Primary

(2) Factor

(3) Term

(4) Arithmetic expression

The formation rules to be applied in establishing the interpretation of arithmetic expressions are in 6.1.2.1 through 6.1.2.4.

6.1.2.1 <u>Primaries</u>. The <u>primaries</u> are:

(1) Unsigned arithmetic constant (4.2.3)

(2) Symbolic name of an arithmetic constant (8.6)

(3) Arithmetic variable reference (2.5)

(4) Arithmetic array element reference (5.3)

(5) Arithmetic function reference (15.2)

(6) Arithmetic expression enclosed in parentheses (6.1.2.4)

6.1.2.2 <u>Factor</u>. The forms of a <u>factor</u> are:

(1) Primary

(2) Primary ** factor

Thus, a factor is formed from a sequence of one or more primaries separated by the exponentiation operator. Form (2) indicates that in interpreting a factor containing two or more exponentiation operators, the primaries are combined from right to left. For example, the factor

```
2**3**2
```

has the same interpretation as the factor

```
2**(3**2)
```

6.1.2.3 <u>Term</u>. The forms of a <u>term</u> are:

(1) Factor

(2) Term / factor

(3) Term * factor

Thus, a term is formed from a sequence of one or more factors separated by either the multiplication operator or the division operator. Forms (2) and (3) indicate that in interpreting a term containing two or more multiplication or division operators, the factors are combined from left to right.

6.1.2.4 Arithmetic Expression. The forms of an arithmetic expression are:

(1) Term

(2) + term

(3) - term

(4) Arithmetic expression + term

(5) Arithmetic expression - term

Thus, an arithmetic expression is formed from a sequence of one or more terms separated by either the addition operator or the subtraction operator. The first term in an arithmetic expression may be preceded by the identity or the negation operator. Forms (4) and (5) indicate that in interpreting an arithmetic expression containing two or more addition or subtraction operators, the terms are combined from left to right.

Note that these formation rules do not permit expressions containing two consecutive arithmetic operators, such as A**-B or A+-B. However, expressions such as A**(-B) and A+(-B) are permitted.

6.1.3 Arithmetic Constant Expression. An arithmetic constant expression is an arithmetic expression in which each primary is an arithmetic constant or an arithmetic constant expression enclosed in parentheses. The exponentiation operator is not permitted unless the exponent is of type integer. Note that variable, array element, and function references are not allowed.

6.1.3.1 Integer Constant Expression. An integer constant expression is an arithmetic constant expression in which each constant is of type integer. Note that variable, array element, and function references are not allowed.

The following are examples of integer constant expressions:

```
3
-3
-3+4
```

6.1.4 Type and Interpretation of Arithmetic Expressions. The data type of a constant is determined by the form of the constant (4.2.1). The data type of an arithmetic variable reference, arithmetic array element reference, or arithmetic interpreting a term containing two or more multiplication or division operators, the factors are combined from left to right.

6.1.2.4 <u>Arithmetic Expression</u>. The forms of an <u>arithmetic expression</u> are:

(1) Term

(2) + term

(3) - term

(4) Arithmetic expression + term

(5) Arithmetic expression - term

Thus, an arithmetic expression is formed from a sequence of one or more terms separated by either the addition operator or the subtraction operator. The first term in an arithmetic expression may be preceded by the identity or the negation operator. Forms (4) and (5) indicate that in interpreting an arithmetic expression containing two or more addition or subtraction operators, the terms are combined from left to right.

Note that these formation rules do not permit expressions containing two consecutive arithmetic operators, such as A**-B or A+-B. However, expressions such as A**(-B) and A+(-B) are permitted.

6.1.3 <u>Arithmetic Constant Expression</u>. An <u>arithmetic constant expression</u> is an arithmetic expression in which each primary is an arithmetic constant, the symbolic name of an arithmetic constant, or an arithmetic constant expression enclosed in parentheses. The exponentiation operator is not permitted unless the exponent is of type integer. Note that variable, array element, and function references are not allowed.

6.1.3.1 <u>Integer Constant Expression</u>. An <u>integer constant expression</u> is an arithmetic constant expression in which each constant or symbolic name of a constant is of type integer. Note that variable, array element, and function references are not allowed.

The following are examples of integer constant expressions:

```
3
-3
-3+4
```

6.1.4 <u>Type and Interpretation of Arithmetic Expressions</u>. The data type of a constant is determined by the form of the constant (4.2.1). The data type of an arithmetic variable reference, symbolic name of an arithmetic constant, function reference is determined by the name of the datum or function (4.1.2). The data type of an arithmetic expression containing one or more arithmetic operators is determined from the data types of the operands.

Integer expressions and real expressions are arithmetic expressions whose values are of type integer and real, respectively.

When the operator + or - operates on a single operand, the data type of the resulting expression is the same as the data type of the operand.

When an arithmetic operator operates on a pair of operands, the data type of the resulting expression is given in Tables 2 and 3. In these tables, each letter I or R represents an operand or result of type integer or real, respectively.

The type of the result is indicated by the I or R that precedes the equals, and the interpretation is indicated by the expression to the right of the equals. REAL is the type-conversion function described in 15.10.

Table 2

Type and Interpretation of Result for $x_1 + x_2$

x_1 \ x_2	I_2	R_2
I_1	$I = I_1 + I_2$	$R = REAL(I_1) + R_2$
R_1	$R = R_1 + REAL(I_2)$	$R = R_1 + R_2$

arithmetic array element reference, or arithmetic function reference is determined by the name of the datum or function (4.1.2). The data type of an arithmetic expression containing one or more arithmetic operators is determined from the data types of the operands.

<u>Integer expressions</u>, <u>real expressions</u>, <u>double precision expressions</u>, and <u>complex expressions</u> are arithmetic expressions whose values are of type integer, real, double precision, and complex, respectively.

When the operator + or - operates on a single operand, the data type of the resulting expression is the same as the data type of the operand.

When an arithmetic operator operates on a pair of operands, the data type of the resulting expression is given in Tables 2 and 3. In these tables, each letter I, R, D, or C represents an operand or result of type integer, real, double precision, or complex, respectively.

The type of the result is indicated by the I, R, D, or C that precedes the equals, and the interpretation is indicated by the expression to the right of the equals. REAL, DBLE, and CMPLX are the type-conversion functions described in 15.10.

<u>Table 2</u>

Type and Interpretation of Result for $x_1 + x_2$

x_1 \ x_2	I_2	R_2
I_1	$I = I_1 + I_2$	$R = REAL(I_1) + R_2$
R_1	$R = R_1 + REAL(I_2)$	$R = R_1 + R_2$
D_1	$D = D_1 + DBLE(I_2)$	$D = D_1 + DBLE(R_2)$
C_1	$C=C_1+CMPLX(REAL(I_2),0.)$	$C = C_1 + CMPLX(R_2,0.)$

x_1 \ x_2	D_2	C_2
I_1	$D = DBLE(I_1) + D_2$	$C=CMPLX(REAL(I_1),0.)+C_2$
R_1	$D = DBLE(R_1) + D_2$	$C = CMPLX(R_1,0.) + C_2$
D_1	$D = D_1 + D_2$	Prohibited
C_1	Prohibited	$C = C_1 + C_2$

Tables giving the type and interpretation of expressions involving -, *, and / may be obtained by replacing all occurrences of + in Table 2 by -, *, or /, respectively.

Table 3

Type and Interpretation of Result for x_1**x_2

x_2 / x_1	I_2	R_2
I_1	I = I_1**I_2	R = REAL(I_1)**R_2
R_1	R = R_1**I_2	R = R_1**R_2

Except for a value raised to an integer power, Tables 2 and 3 specify that if two operands are of different type, the operand that differs in type from the result of the operation is converted to the type of the result and then the operator operates on a pair of operands of the same type. When a primary of type real is raised to an integer power, the integer operand need not be converted. If the value of I_2 is negative, the interpretation of I_1**I_2 is the same as the interpretation of 1/(I_1**IABS(I_2)), which is subject to the rules for integer division (6.1.5). For example, 2**(-3) has the value of 1/(2**3), which is zero.

The type and interpretation of an expression that consists of an operator operating on either a single operand or a pair of operands are independent of the context in which the

Tables giving the type and interpretation of expressions involving -, *, and / may be obtained by replacing all occurrences of + in Table 2 by -, *, or /, respectively.

Table 3

Type and Interpretation of Result for x_1**x_2

x_2 / x_1	I_2	R_2
I_1	$I = I_1$**I_2	R = REAL(I_1)**R_2
R_1	$R = R_1$**I_2	$R = R_1$**R_2
D_1	$D = D_1$**I_2	$D = D_1$**DBLE(R_2)
C_1	$C = C_1$**I_2	$C = C_1$**CMPLX(R_2,0.)

x_2 / x_1	D_2	C_2
I_1	D = DBLE(I_1)**D_2	C=CMPLX(REAL(I_1),0.)**C_2
R_1	D = DBLE(R_1)**D_2	C = CMPLX(R_1,0.)**C_2
D_1	$D = D_1$**D_2	Prohibited
C_1	Prohibited	$C = C_1$**C_2

Four entries in Table 3 specify an interpretation to be a complex value raised to a complex power. In these cases, the value of the expression is the "principal value" determined by x_1**x_2 = EXP(x_2*LOG(x_1)), where EXP and LOG are functions described in 15.10.

Except for a value raised to an integer power, Tables 2 and 3 specify that if two operands are of different type, the operand that differs in type from the result of the operation is converted to the type of the result and then the operator operates on a pair of operands of the same type. When a primary of type real, double precision, or complex is raised to an integer power, the integer operand need not be converted. If the value of I_2 is negative, the interpretation of I_1**I_2 is the same as the interpretation of 1/(I_1**ABS(I_2)), which is subject to the rules for integer division (6.1.5). For example, 2**(-3) has the value of 1/(2**3), which is zero.

The type and interpretation of an expression that consists of an operator operating on either a single operand or a pair of operands are independent of the context in which the expression appears. In particular, the type and interpretation of such an expression are independent of the type of any other operand of any larger expression in which it appears. For example, if X is of type real, J is of type integer, and INT is the real-to-integer conversion function, the expression INT(X+J) is an integer expression and X+J is a real expression.

6.1.5 Integer Division. One operand of type integer may be divided by another operand of type integer. Although the mathematical quotient of two integers is not necessarily an integer, Table 2 specifies that an expression involving the division operator with two operands of type integer is interpreted as an expression of type integer. The result of such a division is called an integer quotient and is obtained as follows: If the magnitude of the mathematical quotient is less than one, the integer quotient is zero. Otherwise, the integer quotient is the integer whose magnitude is the largest integer that does not exceed the magnitude of the mathematical quotient and whose sign is the same as the sign of the mathematical quotient. For example, the value of the expression (-8)/3 is (-2).

6.2 Character Expressions

A character expression is used to express a character string. Evaluation of a character expression produces a result of type character.

The simplest form of a character expression is a character constant, character variable reference, or character array element reference.

6.2.1 Character Operator. The concatenation operator is not included in the subset.

expression appears. In particular, the type and interpretation of such an expression are independent of the type of any other operand of any larger expression in which it appears. For example, if X is of type real, J is of type integer, and INT is the real-to-integer conversion function, the expression INT(X+J) is an integer expression and X+J is a real expression.

6.1.5 Integer Division. One operand of type integer may be divided by another operand of type integer. Although the mathematical quotient of two integers is not necessarily an integer, Table 2 specifies that an expression involving the division operator with two operands of type integer is interpreted as an expression of type integer. The result of such a division is called an integer quotient and is obtained as follows: If the magnitude of the mathematical quotient is less than one, the integer quotient is zero. Otherwise, the integer quotient is the integer whose magnitude is the largest integer that does not exceed the magnitude of the mathematical quotient and whose sign is the same as the sign of the mathematical quotient. For example, the value of the expression (-8)/3 is (-2).

6.2 Character Expressions

A character expression is used to express a character string. Evaluation of a character expression produces a result of type character.

The simplest form of a character expression is a character constant, symbolic name of a character constant, character variable reference, character array element reference, character substring reference, or character function reference. More complicated character expressions may be formed by using one or more character operands together with character operators and parentheses.

6.2.1 Character Operator. The character operator is:

Operator	Representing
//	Concatenation

The interpretation of the expression formed with the character operator is:

Use of Operator	Interpretation
x_1 // x_2	Concatenate x_1 with x_2

where: x_1 denotes the operand to the left of the operator

6.2.2 Form and Interpretation of Character Expressions. A character expression must identify a value of type character.

6.2.2.1 Character Primaries. The character primaries are:

(1) Character constant (4.8.1)

(2) Character variable reference (2.5)

(3) Character array element reference (5.3)

(4) Character expression enclosed in parentheses (6.2.2.2)

6.2.2.2 Character Expression. The form of a character expression is:

(1) Character primary

x_2 denotes the operand to the right of the operator

The result of a concatenation operation is a character string whose value is the value of x_1 concatenated on the right with the value of x_2 and whose length is the sum of the lengths of x_1 and x_2. For example, the value of 'AB' // 'CDE' is the string ABCDE.

6.2.2 Form and Interpretation of Character Expressions. A character expression and the operands of a character expression must identify values of type character. Except in a character assignment statement (10.4), a character expression must not involve concatenation of an operand whose length specification is an asterisk in parentheses (8.4.2) unless the operand is the symbolic name of a constant.

6.2.2.1 Character Primaries. The *character primaries* are:

(1) Character constant (4.8.1)

(2) Symbolic name of a character constant (8.6)

(3) Character variable reference (2.5)

(4) Character array element reference (5.3)

(5) Character substring reference (5.7)

(6) Character function reference (15.2)

(7) Character expression enclosed in parentheses (6.2.2.2)

6.2.2.2 Character Expression. The forms of a *character expression* are:

(1) Character primary

(2) Character expression // character primary

Thus, a character expression is a sequence of one or more character primaries separated by the concatenation operator. Form (2) indicates that in a character expression containing two or more concatenation operators, the primaries are combined from left to right to establish the interpretation of the expression. For example, the formation rules specify that the interpretation of the character expression

```
'AB' // 'CD' // 'EF'
```

is the same as the interpretation of the character expression

```
('AB' // 'CD') // 'EF'
```

Parentheses have no effect on the value of a character expression.

6.2.3 Character Constant Expression. A character constant expression is a character expression in which each primary is a character constant or a character constant expression enclosed in parentheses. Note that variable, array element, and function references are not allowed.

6.3 Relational Expressions

A relational expression is used to compare the values of two arithmetic expressions or two character expressions. A relational expression may not be used to compare the value of an arithmetic expression with the value of a character expression.

Relational expressions may appear only within logical expressions. Evaluation of a relational expression produces a result of type logical, with a value of true or false.

6.3.1 Relational Operators. The relational operators are:

Operator	Representing
.LT.	Less than
.LE.	Less than or equal to
.EQ.	Equal to
.NE.	Not equal to
.GT.	Greater than
.GE.	Greater than or equal to

6.3.2 Arithmetic Relational Expression. The form of an arithmetic relational expression is:

$$e_1 \; relop \; e_2$$

where: e_1 and e_2 are each an integer or real expression

relop is a relational operator

6.3.3 Interpretation of Arithmetic Relational Expressions. An arithmetic relational expression is interpreted as having the logical value true if the values of the operands satisfy the relation specified by the operator. An arithmetic relational expression is interpreted as having the logical

The value of the character expression in this example is the same as that of the constant 'ABCDEF'. Note that parentheses have no effect on the value of a character expression.

6.2.3 Character Constant Expression. A character constant expression is a character expression in which each primary is a character constant, the symbolic name of a character constant, or a character constant expression enclosed in parentheses. Note that variable, array element, substring, and function references are not allowed.

6.3 Relational Expressions

A relational expression is used to compare the values of two arithmetic expressions or two character expressions. A relational expression may not be used to compare the value of an arithmetic expression with the value of a character expression.

Relational expressions may appear only within logical expressions. Evaluation of a relational expression produces a result of type logical, with a value of true or false.

6.3.1 Relational Operators. The relational operators are:

Operator	Representing
.LT.	Less than
.LE.	Less than or equal to
.EQ.	Equal to
.NE.	Not equal to
.GT.	Greater than
.GE.	Greater than or equal to

6.3.2 Arithmetic Relational Expression. The form of an arithmetic relational expression is:

$\underline{e_1}$ $\underline{relop}$ $\underline{e_2}$

where: $\underline{e_1}$ and $\underline{e_2}$ are each an integer, real, double precision, or complex expression

relop is a relational operator

A complex operand is permitted only when the relational operator is .EQ. or .NE.

6.3.3 Interpretation of Arithmetic Relational Expressions. An arithmetic relational expression is interpreted as having the logical value true if the values of the operands satisfy the relation specified by the operator. An arithmetic relational expression is interpreted as having the logical value false if the values of the operands do not satisfy the relation specified by the operator.

If the two arithmetic expressions are of different types, the value of the relational expression

$\underline{e}_1$ relop $\underline{e}_2$

is the value of the expression

$((\underline{e}_1) - (\underline{e}_2))$ relop 0

where 0 (zero) is of the same type as the expression $((\underline{e}_1)-(\underline{e}_2))$, and relop is the same relational operator in both expressions.

6.3.4 Character Relational Expression. The form of a character relational expression is:

$\underline{e}_1$ relop $\underline{e}_2$

where: $\underline{e}_1$ and $\underline{e}_2$ are character expressions

relop is a relational operator

6.3.5 Interpretation of Character Relational Expressions. A character relational expression is interpreted as the logical value true if the values of the operands satisfy the relation specified by the operator. A character relational expression is interpreted as the logical value false if the values of the operands do not satisfy the relation specified by the operator.

The character expression $\underline{e}_1$ is considered to be less than $\underline{e}_2$ if the value of $\underline{e}_1$ precedes the value of $\underline{e}_2$ in the collating sequence; $\underline{e}_1$ is greater than $\underline{e}_2$ if the value of $\underline{e}_1$ follows the value of $\underline{e}_2$ in the collating sequence (3.1.5). Note that the collating sequence depends partially on the processor; however, the result of the use of the operators .EQ. and .NE. does not depend on the collating sequence. If the operands are of unequal length, the shorter operand is considered as if it were extended on the right with blanks to the length of the longer operand.

6.4 Logical Expressions

A logical expression is used to express a logical computation. Evaluation of a logical expression produces a result of type logical, with a value of true or false.

The simplest form of a logical expression is a logical constant, logical variable reference, logical array element reference, logical function reference, or relational expression. More complicated logical expressions may be value false if the values of the operands do not satisfy the relation specified by the operator.

If the two arithmetic expressions are of different types, the value of the relational expression

$$\underline{e}_1 \ \underline{relop} \ \underline{e}_2$$

is the value of the expression

$$((\underline{e}_1) - (\underline{e}_2)) \ \underline{relop} \ 0$$

where 0 (zero) is of the same type as the expression $((\underline{e}_1)-(\underline{e}_2))$, and *relop* is the same relational operator in both expressions. Note that the comparison of a double precision value and a complex value is not permitted.

6.3.4 *Character Relational Expression*. The form of a *character relational expression* is:

$$\underline{e}_1 \ \underline{relop} \ \underline{e}_2$$

where: $\underline{e}_1$ and $\underline{e}_2$ are character expressions

relop is a relational operator

6.3.5 *Interpretation of Character Relational Expressions*. A character relational expression is interpreted as the logical value true if the values of the operands satisfy the relation specified by the operator. A character relational expression is interpreted as the logical value false if the values of the operands do not satisfy the relation specified by the operator.

The character expression $\underline{e}_1$ is considered to be less than $\underline{e}_2$ if the value of $\underline{e}_1$ precedes the value of $\underline{e}_2$ in the collating sequence; $\underline{e}_1$ is greater than $\underline{e}_2$ if the value of $\underline{e}_1$ follows the value of $\underline{e}_2$ in the collating sequence (3.1.5). Note that the collating sequence depends partially on the processor; however, the result of the use of the operators .EQ. and .NE. does not depend on the collating sequence. If the operands are of unequal length, the shorter operand is considered as if it were extended on the right with blanks to the length of the longer operand.

6.4 *Logical Expressions*

A logical expression is used to express a logical computation. Evaluation of a logical expression produces a result of type logical, with a value of true or false.

The simplest form of a logical expression is a logical constant, symbolic name of a logical constant, logical variable reference, logical array element reference, logical function reference, or relational expression. More complicated logical expressions may be formed by using one formed by using one or more logical operands together with logical operators and parentheses.

6.4.1 Logical Operators. The logical operators are:

Operator	Representing
.NOT.	Logical Negation
.AND.	Logical Conjunction
.OR.	Logical Inclusive Disjunction

6.4.2 Form and Interpretation of Logical Expressions. A set of formation rules is used to establish the interpretation of a logical expression that contains two or more logical operators. There is a precedence among the logical operators, which determines the order in which the operands are to be combined unless the order is changed by the use of parentheses. The precedence of the logical operators is as follows:

Operator	Precedence
.NOT.	Highest
.AND.	
.OR.	Lowest

For example, in the expression

    A .OR. B .AND. C

the .AND. operator has higher precedence than the .OR. operator; therefore, the interpretation of the above expression is the same as the interpretation of the expression

    A .OR. (B .AND. C)

The logical operands are:

(1) Logical primary

(2) Logical factor

(3) Logical term

(4) Logical disjunct

(5) Logical expression

or more logical operands together with logical operators and parentheses.

6.4.1 Logical Operators. The logical operators are:

Operator	Representing
.NOT.	Logical Negation
.AND.	Logical Conjunction
.OR.	Logical Inclusive Disjunction
.EQV.	Logical Equivalence
.NEQV.	Logical Nonequivalence

6.4.2 Form and Interpretation of Logical Expressions. A set of formation rules is used to establish the interpretation of a logical expression that contains two or more logical operators. There is a precedence among the logical operators, which determines the order in which the operands are to be combined unless the order is changed by the use of parentheses. The precedence of the logical operators is as follows:

Operator	Precedence
.NOT.	Highest
.AND.	
.OR.	
.EQV. or .NEQV.	Lowest

For example, in the expression

```
A .OR. B .AND. C
```

the .AND. operator has higher precedence than the .OR. operator; therefore, the interpretation of the above expression is the same as the interpretation of the expression

```
A .OR. (B .AND. C)
```

The logical operands are:

(1) Logical primary

(2) Logical factor

(3) Logical term

(4) Logical disjunct

(5) Logical expression

The formation rules to be applied in establishing the interpretation of a logical expression are in 6.4.2.1 through 6.4.2.5.

6.4.2.1 Logical Primaries. The logical primaries are:

(1) Logical constant (4.7.1)

(2) Logical variable reference (2.5)

(3) Logical array element reference (5.3)

(4) Logical function reference (15.2)

(5) Relational expression (6.3)

(6) Logical expression enclosed in parentheses (6.4.2.5)

6.4.2.2 Logical Factor. The forms of a logical factor are:

(1) Logical primary

(2) .NOT. logical primary

6.4.2.3 Logical Term. The forms of a logical term are:

(1) Logical factor

(2) Logical term .AND. logical factor

Thus, a logical term is a sequence of logical factors separated by the .AND. operator. Form (2) indicates that in interpreting a logical term containing two or more .AND. operators, the logical factors are combined from left to right.

6.4.2.4 Logical Disjunct. The forms of a logical disjunct are:

(1) Logical term

(2) Logical disjunct .OR. logical term

Thus, a logical disjunct is a sequence of logical terms separated by the .OR. operator. Form (2) indicates that in interpreting a logical disjunct containing two or more .OR. operators, the logical terms are combined from left to right.

The formation rules to be applied in establishing the interpretation of a logical expression are in 6.4.2.1 through 6.4.2.5.

6.4.2.1 Logical Primaries. The logical primaries are:

(1) Logical constant (4.7.1)

(2) Symbolic name of a logical constant (8.6)

(3) Logical variable reference (2.5)

(4) Logical array element reference (5.3)

(5) Logical function reference (15.2)

(6) Relational expression (6.3)

(7) Logical expression enclosed in parentheses (6.4.2.5)

6.4.2.2 Logical Factor. The forms of a logical factor are:

(1) Logical primary

(2) .NOT. logical primary

6.4.2.3 Logical Term. The forms of a logical term are:

(1) Logical factor

(2) Logical term .AND. logical factor

Thus, a logical term is a sequence of logical factors separated by the .AND. operator. Form (2) indicates that in interpreting a logical term containing two or more .AND. operators, the logical factors are combined from left to right.

6.4.2.4 Logical Disjunct. The forms of a logical disjunct are:

(1) Logical term

(2) Logical disjunct .OR. logical term

Thus, a logical disjunct is a sequence of logical terms separated by the .OR. operator. Form (2) indicates that in interpreting a logical disjunct containing two or more .OR. operators, the logical terms are combined from left to right.

6.4.2.5 Logical Expression. The form of a logical expression is:

(1) Logical disjunct

The logical equivalence operators, ,EQV. and .NEQV., are not included in the subset.

6.4.3 Value of Logical Factors, Terms, Disjuncts, and Expressions. The value of a logical factor involving .NOT. is shown below:

x_2	.NOT. x_2
true	false
false	true

The value of a logical term involving .AND. is shown below:

x_1	x_2	x_1 .AND. x_2
true	true	true
true	false	false
false	true	false
false	false	false

The value of a logical disjunct involving .OR. is shown below:

x_1	x_2	x_1 .OR. x_2
true	true	true
true	false	true
false	true	true
false	false	false

6.4.2.5 Logical Expression. The forms of a logical expression are:

(1) Logical disjunct

(2) Logical expression .EQV. logical disjunct

(3) Logical expression .NEQV. logical disjunct

Thus, a logical expression is a sequence of logical disjuncts separated by either the .EQV. operator or the .NEQV. operator. Forms (2) and (3) indicate that in interpreting a logical expression containing two or more .EQV. or .NEQV. operators, the logical disjuncts are combined from left to right.

6.4.3 Value of Logical Factors, Terms, Disjuncts, and Expressions. The value of a logical factor involving .NOT. is shown below:

x_2	.NOT. x_2
true	false
false	true

The value of a logical term involving .AND. is shown below:

x_1	x_2	x_1 .AND. x_2
true	true	true
true	false	false
false	true	false
false	false	false

The value of a logical disjunct involving .OR. is shown below:

x_1	x_2	x_1 .OR. x_2
true	true	true
true	false	true
false	true	true
false	false	false

6.4.4 Logical Constant Expression. A logical constant expression is a logical expression in which each primary is a logical constant, a relational expression in which each primary is a constant expression, or a logical constant expression enclosed in parentheses. Note that variable, array element, and function references are not allowed.

6.5 Precedence of Operators

In 6.1.2 and 6.4.2 precedences have been established among the arithmetic operators and the logical operators, respectively. No precedence has been established among the relational operators. The precedences among the various operators are:

Operator	Precedence
Arithmetic Relational Logical	Highest Lowest

An expression may contain more than one kind of operator. For example, the logical expression

```
L .OR. A + B .GE. C
```

where A, B, and C are of type real, and L is of type logical, contains an arithmetic operator, a relational

The value of a logical expression involving .EQV. is shown below:

x_1	x_2	x_1 .EQV. x_2
true	true	true
true	false	false
false	true	false
false	false	true

The value of a logical expression involving .NEQV. is shown below:

x_1	x_2	x_1 .NEQV. x_2
true	true	false
true	false	true
false	true	true
false	false	false

6.4.4 Logical Constant Expression. A logical constant expression is a logical expression in which each primary is a logical constant, the symbolic name of a logical constant, a relational expression in which each primary is a constant expression, or a logical constant expression enclosed in parentheses. Note that variable, array element, and function references are not allowed.

6.5 Precedence of Operators

In 6.1.2 and 6.4.2 precedences have been established among the arithmetic operators and the logical operators, respectively. There is only one character operator. No precedence has been established among the relational operators. The precedences among the various operators are:

Operator	Precedence
Arithmetic	Highest
Character	
Relational	
Logical	Lowest

An expression may contain more than one kind of operator. For example, the logical expression

```
L .OR. A + B .GE. C
```

where A, B, and C are of type real, and L is of type logical, contains an arithmetic operator, a relational operator, and a logical operator. This expression would be interpreted the same as the expression

```
L .OR. ((A + B) .GE. C)
```

6.5.1 Summary of Interpretation Rules. The order in which primaries are combined using operators is determined by the following:

(1) Use of parentheses

(2) Precedence of the operators

(3) Right-to-left interpretation of exponentiations in a factor

(4) Left-to-right interpretation of multiplications and divisions in a term

(5) Left-to-right interpretation of add tions and subtractions in an arithmetic expression

(6) Left-to-right interpretation of conjunctions in a logical term

(7) Left-to-right interpretation of disjunctions in a logical disjunct

6.6 Evaluation of Expressions

This section applies to arithmetic, character, relational, and logical expressions.

Any variable, array element, or function referenced as an operand in an expression must be defined at the time the reference is executed. An integer operand must be defined with an integer value rather than a statement label value. Note that if a character string is referenced, all of the referenced characters must be defined at the time the reference is executed.

Any arithmetic operation whose result is not mathematically defined is prohibited in the execution of an executable program. Examples are dividing by zero and raising a zero-valued primary to a zero-valued or negative-valued power. Raising a negative-valued primary to a real power is also prohibited.

The execution of a function reference in a statement may not alter the value of any other entity within the statement in operator, and a logical operator. This expression would be interpreted the same as the expression

```
L .OR. ((A + B) .GE. C)
```

6.5.1 Summary of Interpretation Rules. The order in which primaries are combined using operators is determined by the following:

(1) Use of parentheses

(2) Precedence of the operators

(3) Right-to-left interpretation of exponentiations in a factor

(4) Left-to-right interpretation of multiplications and divisions in a term

(5) Left-to-right interpretation of additions and subtractions in an arithmetic expression

(6) Left-to-right interpretation of concatenations in a character expression

(7) Left-to-right interpretation of conjunctions in a logical term

(8) Left-to-right interpretation of disjunctions in a logical disjunct

(9) Left-to-right interpretation of logical equivalences in a logical expression

6.6 Evaluation of Expressions

This section applies to arithmetic, character, relational, and logical expressions.

Any variable, array element, function, or character substring referenced as an operand in an expression must be defined at the time the reference is executed. An integer operand must be defined with an integer value rather than a statement label value. Note that if a character string or substring is referenced, all of the referenced characters must be defined at the time the reference is executed.

Any arithmetic operation whose result is not mathematically defined is prohibited in the execution of an executable program. Examples are dividing by zero and raising a zero-valued primary to a zero-valued or negative-valued power. Raising a negative-valued primary to a real or double precision power is also prohibited.

The execution of a function reference in a statement may not alter the value of any other entity within the statement in which the function reference appears. The execution of a function reference in a statement may not alter the value of any entity in common (8.3) that affects the value of any other function reference in that statement. However, execution of a function reference in the expression _e_ of a logical IF statement (11.5) is permitted to affect entities in the statement _st_ that is executed when the value of the expression _e_ is true. If a function reference causes definition of an actual argument of the function, that argument or any associated entities must not appear elsewhere in the same statement. For example, the statements

```
A(I) = F(I)

Y = G(X) + X
```

are prohibited if the reference to F defines I or the reference to G defines X.

The data type of an expression in which a function reference appears does not affect the evaluation of the actual arguments of the function. The data type of an expression in which a function reference appears is not affected by the evaluation of the actual arguments of the function.

Any execution of an array element reference requires the evaluation of its subscript. The data type of an expression in which a subscript appears does not affect, nor is it affected by, the evaluation of the subscript.

6.6.1 Evaluation of Operands. It is not necessary for a processor to evaluate all of the operands of an expression if the value of the expression can be determined otherwise. This principle is most often applicable to logical expressions, but it applies to all expressions. For example, in evaluating the logical expression

```
X .GT. Y .OR. L(Z)
```

where X, Y, and Z are real, and L is a logical function, the function reference L(Z) need not be evaluated if X is greater than Y. If a statement contains a function reference in a part of an expression that need not be evaluated, all entities that would have become defined in the execution of that reference become undefined at the completion of evaluation of the expression containing the function reference. In the example above, evaluation of the which the function reference appears. The execution of a function reference in a statement may not alter the value of any entity in common (8.3) that affects the value of any other function reference in that statement. However, execution of a function reference in the expression _e_ of a logical IF statement (11.5) is permitted to affect entities in the statement _st_ that is executed when the value of the expression _e_ is true. If a function reference causes definition of an actual argument of the function, that argument or any associated entities must not appear elsewhere in the same statement. For example, the statements

```
A(I) = F(I)

Y = G(X) + X
```

are prohibited if the reference to F defines I or the reference to G defines X.

The data type of an expression in which a function reference appears does not affect the evaluation of the actual arguments of the function. The data type of an expression in which a function reference appears is not affected by the evaluation of the actual arguments of the function, except that the result of a generic function reference assumes a data type that depends on the data type of its arguments as specified in 15.10.

Any execution of an array element reference requires the evaluation of its subscript. The data type of an expression in which a subscript appears does not affect, nor is it affected by, the evaluation of the subscript.

Any execution of a substring reference requires the evaluation of its substring expressions. The data type of an expression in which a substring name appears does not affect, nor is it affected by, the evaluation of the substring expressions.

6.6.1 _Evaluation of Operands_. It is not necessary for a processor to evaluate all of the operands of an expression if the value of the expression can be determined otherwise. This principle is most often applicable to logical expressions, but it applies to all expressions. For example, in evaluating the logical expression

```
X .GT. Y .OR. L(Z)
```

where X, Y, and Z are real, and L is a logical function, the function reference L(Z) need not be evaluated if X is greater than Y. If a statement contains a function reference in a part of an expression that need not be evaluated, all entities that would have become defined in the execution of that reference become undefined at the completion of evaluation of the expression containing the function reference. In the example above, evaluation of the expression causes Z to become undefined if L defines its argument.

6.6.2 Order of Evaluation of Functions. If a statement contains more than one function reference, a processor may evaluate the functions in any order, except for a logical IF statement and a function argument list containing function references. For example, the statement

```
Y = F(G(X))
```

where F and G are functions, requires G to be evaluated before F is evaluated.

In a statement that contains more than one function reference, the value provided by each function reference must be independent of the order chosen by the processor for evaluation of the function references.

6.6.3 Integrity of Parentheses. The sections that follow state certain conditions under which a processor may evaluate an expression different from the one obtained by applying the interpretation rules given in 6.1 through 6.5. However, any expression contained in parentheses must be treated as an entity. For example, in evaluating the expression A*(B*C), the product of B and C must be evaluated and then multiplied by A; the processor must not evaluate the mathematically equivalent expression (A*B)*C.

6.6.4 Evaluation of Arithmetic Expressions. The rules given in 6.1.2 specify the interpretation of an arithmetic expression. Once the interpretation has been established in accordance with those rules, the processor may evaluate any mathematically equivalent expression, provided that the integrity of parentheses is not violated.

Two arithmetic expressions are mathematically equivalent if, for all possible values of their primaries, their mathematical values are equal. However, mathematically equivalent arithmetic expressions may produce different computational results.

The mathematical definition of integer division is given in 6.1.5. The difference between the value of the expression 5/2 and 5./2. is a mathematical difference, not a computational difference.

The following are examples of expressions, along with allowable alternative forms that may be used by the processor in the evaluation of those expressions. A, B, and C represent arbitrary real operands; I and J represent arbitrary integer operands; and X, Y, and Z represent arbitrary arithmetic operands.

expression causes Z to become undefined if L defines its argument.

6.6.2 Order of Evaluation of Functions. If a statement contains more than one function reference, a processor may evaluate the functions in any order, except for a logical IF statement and a function argument list containing function references. For example, the statement

```
Y = F(G(X))
```

where F and G are functions, requires G to be evaluated before F is evaluated.

In a statement that contains more than one function reference, the value provided by each function reference must be independent of the order chosen by the processor for evaluation of the function references.

6.6.3 Integrity of Parentheses. The sections that follow state certain conditions under which a processor may evaluate an expression different from the one obtained by applying the interpretation rules given in 6.1 through 6.5. However, any expression contained in parentheses must be treated as an entity. For example, in evaluating the expression A*(B*C), the product of B and C must be evaluated and then multiplied by A; the processor must not evaluate the mathematically equivalent expression (A*B)*C.

6.6.4 Evaluation of Arithmetic Expressions. The rules given in 6.1.2 specify the interpretation of an arithmetic expression. Once the interpretation has been established in accordance with those rules, the processor may evaluate any mathematically equivalent expression, provided that the integrity of parentheses is not violated.

Two arithmetic expressions are mathematically equivalent if, for all possible values of their primaries, their mathematical values are equal. However, mathematically equivalent arithmetic expressions may produce different computational results.

The mathematical definition of integer division is given in 6.1.5. The difference between the value of the expression 5/2 and 5./2. is a mathematical difference, not a computational difference.

The following are examples of expressions, along with allowable alternative forms that may be used by the processor in the evaluation of those expressions. A, B, and C represent arbitrary real, double precision, or complex operands; I and J represent arbitrary integer operands; and X, Y, and Z represent arbitrary arithmetic operands. (Note that Table 2 prohibits combinations of double precision and complex data types.)

Expression	Allowable Alternative Form
X+Y	Y+X
X*Y	Y*X
-X+Y	Y-X
X+Y+Z	X+(Y+Z)
X-Y+Z	X-(Y-Z)
X*B/Z	X*(B/Z)
X*Y-X*Z	X*(Y-Z)
A/B/C	A/(B*C)
A/5.0	0.2*A

The following are examples of expressions along with forbidden forms that must not be used by the processor in the evaluation of those expressions.

Expression	Nonallowable Alternative Form
I/2	0.5*I
X*I/J	X*(I/J)
I/J/A	I/(J*A)
(X*Y)-(X*Z)	X*(Y-Z)
X*(Y-Z)	X*Y-X*Z

In addition to the parentheses required to establish the desired interpretation, parentheses may be included to restrict the alternative forms that may be used by the processor in the actual evaluation of the expression. This is useful for controlling the magnitude and accuracy of intermediate values developed during the evaluation of an expression. For example, in the expression

```
A+(B-C)
```

the term (B-C) must be evaluated and then added to A. Note that the inclusion of parentheses may change the mathematical value of an expression. For example, the two expressions:

```
A*I/J

A*(I/J)
```

may have different mathematical values if I and J are factors of integer data type.

Each operand of an arithmetic operator has a data type that may depend on the order of evaluation used by the processor. For example, in the evaluation of the expression

```
J+R+I
```

Expression	Allowable Alternative Form
X+Y	Y+X
X*Y	Y*X
-X+Y	Y-X
X+Y+Z	X+(Y+Z)
X-Y+Z	X-(Y-Z)
X*B/Z	X*(B/Z)
X*Y-X*Z	X*(Y-Z)
A/B/C	A/(B*C)
A/5.0	0.2*A

The following are examples of expressions along with forbidden forms that must not be used by the processor in the evaluation of those expressions.

Expression	Nonallowable Alternative Form
I/2	0.5*I
X*I/J	X*(I/J)
I/J/A	I/(J*A)
(X*Y)-(X*Z)	X*(Y-Z)
X*(Y-Z)	X*Y-X*Z

In addition to the parentheses required to establish the desired interpretation, parentheses may be included to restrict the alternative forms that may be used by the processor in the actual evaluation of the expression. This is useful for controlling the magnitude and accuracy of intermediate values developed during the evaluation of an expression. For example, in the expression

```
A+(B-C)
```

the term (B-C) must be evaluated and then added to A. Note that the inclusion of parentheses may change the mathematical value of an expression. For example, the two expressions:

```
A*I/J

A*(I/J)
```

may have different mathematical values if I and J are factors of integer data type.

Each operand of an arithmetic operator has a data type that may depend on the order of evaluation used by the processor. For example, in the evaluation of the expression

```
D+R+I
```

where J, R, and I represent terms of integer, real, and integer data type, respectively, the data type of the operand that is added to I may be either integer or real, depending on which pair of operands (J and R, R and I, or J and I) is added first.

6.6.5 <u>Evaluation of Character Expressions</u>. The rules given in 6.2.2 specify the interpretation of a character expression as a string of characters. A processor needs to evaluate only as much of the character expression as is required by the context in which the expression appears.

6.6.6 <u>Evaluation of Relational Expressions</u>. The rules given in 6.3.3 and 6.3.5 specify the interpretation of relational expressions. Once the interpretation of an expression has been established in accordance with those rules, the processor may evaluate any other expression that is relationally equivalent. For example, the processor may choose to evaluate the relational expression

```
I .GT. J
```

where I and J are integer variables, as

```
J - I .LT. 0
```

Two relational expressions are relationally equivalent if their logical values are equal for all possible values of their primaries.

6.6.7 <u>Evaluation of Logical Expressions</u>. The rules given in 6.4.2 specify the interpretation of a logical expression. Once the interpretation of an expression has been established in accordance with those rules, the processor may evaluate any other expression that is logically equivalent, provided that the integrity of parentheses is not violated. For example, the processor may choose to evaluate the logical expression

```
L1 .AND. L2 .AND. L3
```

where L1, L2, and L3 are logical variables, as

```
L1 .AND. (L2 .AND. L3)
```

Two logical expressions are logically equivalent if their values are equal for all possible values of their primaries.

where D, R, and I represent terms of double precision, real, and integer data type, respectively, the data type of the operand that is added to I may be either double precision or real, depending on which pair of operands (D and R, R and I, or D and I) is added first.

6.6.5 Evaluation of Character Expressions. The rules given in 6.2.2 specify the interpretation of a character expression as a string of characters. A processor needs to evaluate only as much of the character expression as is required by the context in which the expression appears. For example, the statements

```
CHARACTER*2 C1,C2,C3,CF
C1 = C2 // CF(C3)
```

do not require the function CF to be evaluated, because only the value of C2 is needed to determine the value of C1.

6.6.6 Evaluation of Relational Expressions. The rules given in 6.3.3 and 6.3.5 specify the interpretation of relational expressions. Once the interpretation of an expression has been established in accordance with those rules, the processor may evaluate any other expression that is relationally equivalent. For example, the processor may choose to evaluate the relational expression

```
I .GT. J
```

where I and J are integer variables, as

```
J - I .LT. 0
```

Two relational expressions are relationally equivalent if their logical values are equal for all possible values of their primaries.

6.6.7 Evaluation of Logical Expressions. The rules given in 6.4.2 specify the interpretation of a logical expression. Once the interpretation of an expression has been established in accordance with those rules, the processor may evaluate any other expression that is logically equivalent, provided that the integrity of parentheses is not violated. For example, the processor may choose to evaluate the logical expression

```
L1 .AND. L2 .AND. L3
```

where L1, L2, and L3 are logical variables, as

```
L1 .AND. (L2 .AND. L3)
```

Two logical expressions are logically equivalent if their values are equal for all possible values of their primaries.

6.7 Constant Expressions

A constant expression is an arithmetic constant expression (6.1.3), a character constant expression (6.2.3), or a logical constant expression (6.4.4). Constant expressions are defined in the subset but the concept is not used. Certain contexts in the subset require an unsigned or optionally signed constant; however, every context that permits a constant expression, other than an unsigned or optionally signed constant, also permits a general expression.

6.7 Constant Expressions

A <u>constant expression</u> is an arithmetic constant expression (6.1.3), a character constant expression (6.2.3), or a logical constant expression (6.4.4).

7. EXECUTABLE AND NONEXECUTABLE STATEMENT CLASSIFICATION

Each statement is classified as executable or nonexecutable. Executable statements specify actions and form an execution sequence in an executable program. Nonexecutable statements specify characteristics, arrangement, and initial values of data; contain editing information; specify statement functions; and classify program units. Nonexecutable statements are not part of the execution sequence. Nonexecutable statements may be labeled, but such statement labels must not be used to control the execution sequence.

7.1 Executable Statements

The following statements are classified as executable:

(1) Arithmetic, logical, statement label (ASSIGN), and character assignment statements

(2) Unconditional GO TO, assigned GO TO, and computed GO TO statements

(3) Arithmetic IF and logical IF statements

(4) Block IF, ELSE IF, ELSE, and END IF statements

(5) CONTINUE statement

(6) STOP and PAUSE statements

(7) DO statement

(8) READ and WRITE statements

(9) REWIND, BACKSPACE, ENDFILE, and OPEN statements

(10) CALL and RETURN statements

(11) END statement

7.2 Nonexecutable Statements

The following statements are classified as nonexecutable:

(1) PROGRAM, FUNCTION, and SUBROUTINE statements

(2) DIMENSION, COMMON, EQUIVALENCE, IMPLICIT, EXTERNAL, INTRINSIC, and SAVE statements

(3) INTEGER, REAL, LOGICAL, and CHARACTER type-statements

(4) DATA statement

7. EXECUTABLE AND NONEXECUTABLE STATEMENT CLASSIFICATION

Each statement is classified as executable or nonexecutable. Executable statements specify actions and form an execution sequence in an executable program. Nonexecutable statements specify characteristics, arrangement, and initial values of data; contain editing information; specify statement functions; classify program units; and specify entry points within subprograms. Nonexecutable statements are not part of the execution sequence. Nonexecutable statements may be labeled, but such statement labels must not be used to control the execution sequence.

7.1 Executable Statements

The following statements are classified as executable:

(1) Arithmetic, logical, statement label (ASSIGN), and character assignment statements

(2) Unconditional GO TO, assigned GO TO, and computed GO TO statements

(3) Arithmetic IF and logical IF statements

(4) Block IF, ELSE IF, ELSE, and END IF statements

(5) CONTINUE statement

(6) STOP and PAUSE statements

(7) DO statement

(8) READ, WRITE, and PRINT statements

(9) REWIND, BACKSPACE, ENDFILE, OPEN, CLOSE, and INQUIRE statements

(10) CALL and RETURN statements

(11) END statement

7.2 Nonexecutable Statements

The following statements are classified as nonexecutable:

(1) PROGRAM, FUNCTION, SUBROUTINE, ENTRY, and BLOCK DATA statements

(2) DIMENSION, COMMON, EQUIVALENCE, IMPLICIT, PARAMETER, EXTERNAL, INTRINSIC, and SAVE statements

(3) INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, and CHARACTER type-statements

(4) DATA statement

(5) FORMAT statement

(6) Statement function statement

(5) FORMAT statement

(6) Statement function statement

(5) FORMAT statement

(6) Statement function statement

(5) FORMAT statement

(6) Statement function statement

8. SPECIFICATION STATEMENTS

There are eight kinds of specification statements:

(1) DIMENSION

(2) EQUIVALENCE

(3) COMMON

(4) INTEGER, REAL, LOGICAL, and CHARACTER type-statements

(5) IMPLICIT

(6) EXTERNAL

(7) INTRINSIC

(8) SAVE

All specification statements are nonexecutable.

8.1 DIMENSION Statement

A DIMENSION statement is used to specify the symbolic names and dimension specifications of arrays.

The form of a DIMENSION statement is:

DIMENSION a(d) [,a(d)]...

where each a(d) is an array declarator (5.1).

Each symbolic name a appearing in a DIMENSION statement declares a to be an array in that program unit. Note that array declarators may also appear in COMMON statements and type-statements. Only one appearance of a symbolic name as an array name in an array declarator in a program unit is permitted.

8.2 EQUIVALENCE Statement

An EQUIVALENCE statement is used to specify the sharing of storage units by two or more entities in a program unit. This causes association of the entities that share the storage units.

If the equivalenced entities are of different data types, the EQUIVALENCE statement does not cause type conversion or imply mathematical equivalence. If a variable and an array are equivalenced, the variable does not have array properties and the array does not have the properties of a variable.

8. SPECIFICATION STATEMENTS

There are nine kinds of specification statements:

(1) DIMENSION

(2) EQUIVALENCE

(3) COMMON

(4) INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, and CHARACTER type-statements

(5) IMPLICIT

(6) PARAMETER

(7) EXTERNAL

(8) INTRINSIC

(9) SAVE

All specification statements are nonexecutable.

8.1 <u>DIMENSION Statement</u>

A DIMENSION statement is used to specify the symbolic names and dimension specifications of arrays.

The form of a DIMENSION statement is:

DIMENSION <u>a</u>(<u>d</u>) [,<u>a</u>(<u>d</u>)]...

where each <u>a</u>(<u>d</u>) is an array declarator (5.1).

Each symbolic name <u>a</u> appearing in a DIMENSION statement declares <u>a</u> to be an array in that program unit. Note that array declarators may also appear in COMMON statements and type-statements. Only one appearance of a symbolic name as an array name in an array declarator in a program unit is permitted.

8.2 <u>EQUIVALENCE Statement</u>

An EQUIVALENCE statement is used to specify the sharing of storage units by two or more entities in a program unit. This causes association of the entities that share the storage units.

If the equivalenced entities are of different data types, the EQUIVALENCE statement does not cause type conversion or imply mathematical equivalence. If a variable and an array are equivalenced, the variable does not have array properties and the array does not have the properties of a variable.

8.2.1 Form of an EQUIVALENCE Statement. The form of an EQUIVALENCE statement is:

EQUIVALENCE (*nlist*) [,(*nlist*)]...

where each *nlist* is a list (2.10) of variable names, array element names, and array names. Each list must contain at least two names. Names of dummy arguments of an external procedure in a subprogram must not appear in the list. If a variable name is also a function name (15.5.1), that name must not appear in the list.

Each subscript expression in a list *nlist* must be an integer constant.

8.2.2 Equivalence Association. An EQUIVALENCE statement specifies that the storage sequences of the entities whose names appear in a list *nlist* have the same first storage unit. This causes the association of the entities in the list *nlist* and may cause association of other entities (17.1).

8.2.3 Equivalence of Character Entities. An entity of type character may be equivalenced only with other entities of type character. The lengths of the equivalenced entities must be the same.

An EQUIVALENCE statement specifies that the storage sequences of the character entities whose names appear in a list *nlist* have the same first character storage unit. This causes the association of the entities in the list *nlist* and may cause association of other entities (17.1).

8.2.4 Array Names and Array Element Names. If an array element name appears in an EQUIVALENCE statement, the number of subscript expressions must be the same as the number of dimensions specified in the array declarator for the array name.

8.2.1 *Form of an EQUIVALENCE Statement*. The form of an EQUIVALENCE statement is:

EQUIVALENCE (*nlist*) [,(*nlist*)]...

where each *nlist* is a list (2.10) of variable names, array element names, array names, and character substring names. Each list must contain at least two names. Names of dummy arguments of an external procedure in a subprogram must not appear in the list. If a variable name is also a function name (15.5.1), that name must not appear in the list.

Each subscript expression or substring expression in a list *nlist* must be an integer constant expression.

8.2.2 *Equivalence Association*. An EQUIVALENCE statement specifies that the storage sequences of the entities whose names appear in a list *nlist* have the same first storage unit. This causes the association of the entities in the list *nlist* and may cause association of other entities (17.1).

8.2.3 *Equivalence of Character Entities*. An entity of type character may be equivalenced only with other entities of type character. The lengths of the equivalenced entities are not required to be the same.

An EQUIVALENCE statement specifies that the storage sequences of the character entities whose names appear in a list *nlist* have the same first character storage unit. This causes the association of the entities in the list *nlist* and may cause association of other entities (17.1). Any adjacent characters in the associated entities may also have the same character storage unit and thus may also be associated. In the example:

```
CHARACTER A*4, B*4, C(2)*3
EQUIVALENCE (A,C(1)), (B,C(2))
```

the association of A, B, and C can be graphically illustrated as:

```
|01|02|03|04|05|06|07|

|-----A-----|
         |-----B-----|
|--C(1)--|--C(2)--|
```

8.2.4 *Array Names and Array Element Names*. If an array element name appears in an EQUIVALENCE statement, the number of subscript expressions must be the same as the number of dimensions specified in the array declarator for the array name.

The use of an array name unqualified by a subscript in an EQUIVALENCE statement has the same effect as using an array element name that identifies the first element of the array.

8.2.5 Restrictions on EQUIVALENCE Statements. An EQUIVALENCE statement must not specify that the same storage unit is to occur more than once in a storage sequence. For example,

```
DIMENSION A(2)
EQUIVALENCE (A(1),B), (A(2),B)
```

is prohibited, because it would specify the same storage unit for A(1) and A(2). An EQUIVALENCE statement must not specify that consecutive storage units are to be nonconsecutive. For example, the following is prohibited:

```
REAL A(2), R(3)
EQUIVALENCE (A(1),R(1)), (A(2),R(3))
```

8.3 COMMON Statement

The COMMON statement provides a means of associating entities in different program units. This allows different program units to define and reference the same data without using arguments, and to share storage units.

8.3.1 Form of a COMMON Statement. The form of a COMMON statement is:

COMMON [/[cb]/] nlist [[,]/[cb]/ nlist]...

where: cb is a common block name (18.2.1)

nlist is a list (2.10) of variable names, array names, and array declarators. Only one appearance of a symbolic name as a variable name, array name, or array declarator is permitted in all such lists within a program unit. Names of dummy arguments of an external procedure in a subprogram must not appear in the list. If a variable name is also a function name (15.5.1), that name must not appear in the list.

Each omitted cb specifies the blank common block. If the first cb is omitted, the first two slashes are optional.

In each COMMON statement, the entities whose names appear in an nlist following a block name cb are declared to be in common block cb. If the first cb is omitted, all entities whose names appear in the first nlist are specified to be in blank common. Alternatively, the appearance of two slashes with no block name between them declares the entities whose names appear in the list nlist that follows to be in blank common.

The use of an array name unqualified by a subscript in an EQUIVALENCE statement has the same effect as using an array element name that identifies the first element of the array.

8.2.5 Restrictions on EQUIVALENCE Statements. An EQUIVALENCE statement must not specify that the same storage unit is to occur more than once in a storage sequence. For example,

```
DIMENSION A(2)
EQUIVALENCE (A(1),B), (A(2),B)
```

is prohibited, because it would specify the same storage unit for A(1) and A(2). An EQUIVALENCE statement must not specify that consecutive storage units are to be nonconsecutive. For example, the following is prohibited:

```
REAL A(2)
DOUBLE PRECISION D(2)
EQUIVALENCE (A(1),D(1)), (A(2),D(2))
```

8.3 COMMON Statement

The COMMON statement provides a means of associating entities in different program units. This allows different program units to define and reference the same data without using arguments, and to share storage units.

8.3.1 Form of a COMMON Statement. The form of a COMMON statement is:

COMMON [/[cb]/] nlist [[,]/[cb]/ nlist]...

where: cb is a common block name (18.2.1)

nlist is a list (2.10) of variable names, array names, and array declarators. Only one appearance of a symbolic name as a variable name, array name, or array declarator is permitted in all such lists within a program unit. Names of dummy arguments of an external procedure in a subprogram must not appear in the list. If a variable name is also a function name (15.5.1), that name must not appear in the list.

Each omitted cb specifies the blank common block. If the first cb is omitted, the first two slashes are optional.

In each COMMON statement, the entities whose names appear in an nlist following a block name cb are declared to be in common block cb. If the first cb is omitted, all entities whose names appear in the first nlist are specified to be in blank common. Alternatively, the appearance of two slashes with no block name between them declares the entities whose names appear in the list nlist that follows to be in blank common.

Any common block name *cb* or an omitted *cb* for blank common may occur more than once in one or more COMMON statements in a program unit. The list *nlist* following each successive appearance of the same common block name is treated as a continuation of the list for that common block name.

If a character variable or character array is in a common block, all of the entities in that common block must be of type character.

8.3.2 *Common Block Storage Sequence*. For each common block, a *common block storage sequence* is formed as follows:

(1) A storage sequence is formed consisting of the storage sequences of all entities in the lists *nlist* for the common block. The order of the storage sequence is the same as the order of the appearance of the lists *nlist* in the program unit.

(2) The storage sequence formed in (1) is extended to include all storage units of any storage sequence associated with it by equivalence association. The sequence may be extended only by adding storage units beyond the last storage unit. Entities associated with an entity in a common block are considered to be in that common block.

8.3.3 *Size of a Common Block*. The *size of a common block* is the size of its common block storage sequence, including any extensions of the sequence resulting from equivalence association.

Within an executable program, all named common blocks that have the same name must be the same size. Blank common blocks within an executable program are not required to be the same size.

8.3.4 *Common Association*. Within an executable program, the common block storage sequences of all common blocks with the same name have the same first storage unit. Within an executable program, the common block storage sequences of all blank common blocks have the same first storage unit. This results in the association (17.1) of entities in different program units.

8.3.5 *Differences between Named Common and Blank Common*. A blank common block has the same properties as a named common block, except for the following:

(1) Execution of a RETURN or END statement sometimes causes entities in named common blocks to become undefined but never causes entities in blank common to become undefined (15.8.4).

(2) Named common blocks of the same name must be of the same size in all program units of an executable

Any common block name cb or an omitted cb for blank common may occur more than once in one or more COMMON statements in a program unit. The list nlist following each successive appearance of the same common block name is treated as a continuation of the list for that common block name.

If a character variable or character array is in a common block, all of the entities in that common block must be of type character.

8.3.2 Common Block Storage Sequence. For each common block, a common block storage sequence is formed as follows:

(1) A storage sequence is formed consisting of the storage sequences of all entities in the lists nlist for the common block. The order of the storage sequence is the same as the order of the appearance of the lists nlist in the program unit.

(2) The storage sequence formed in (1) is extended to include all storage units of any storage sequence associated with it by equivalence association. The sequence may be extended only by adding storage units beyond the last storage unit. Entities associated with an entity in a common block are considered to be in that common block.

8.3.3 Size of a Common Block. The size of a common block is the size of its common block storage sequence, including any extensions of the sequence resulting from equivalence association.

Within an executable program, all named common blocks that have the same name must be the same size. Blank common blocks within an executable program are not required to be the same size.

8.3.4 Common Association. Within an executable program, the common block storage sequences of all common blocks with the same name have the same first storage unit. Within an executable program, the common block storage sequences of all blank common blocks have the same first storage unit. This results in the association (17.1) of entities in different program units.

8.3.5 Differences between Named Common and Blank Common. A blank common block has the same properties as a named common block, except for the following:

(1) Execution of a RETURN or END statement sometimes causes entities in named common blocks to become undefined but never causes entities in blank common to become undefined (15.8.4).

(2) Named common blocks of the same name must be of the same size in all program units of an executable

program in which they appear, but blank common blocks may be of different sizes.

8.3.6 Restrictions on Common and Equivalence. An EQUIVALENCE statement must not cause the storage sequences of two different common blocks in the same program unit to be associated. Equivalence association must not cause a common block storage sequence to be extended by adding storage units preceding the first storage unit of the first entity specified in a COMMON statement for the common block. For example, the following is not permitted:

```
COMMON /X/A
REAL B(2)
EQUIVALENCE (A,B(2))
```

8.4 Type-Statements

A type-statement is used to override or confirm implicit typing and may specify dimension information.

The appearance of the symbolic name of a variable, array, external function, or statement function in a type-statement specifies the data type for that name for all appearances in the program unit. Within a program unit, a name must not have its type explicitly specified more than once.

A type-statement that confirms the type of an intrinsic function whose name appears in the Specific Name column of Table 5 is not required, but is permitted.

The name of a main program or subroutine must not appear in a type-statement.

8.4.1 INTEGER, REAL, DOUBLE PRECISION, COMPLEX, and LOGICAL Type-Statements.

An INTEGER, REAL, or LOGICAL type-statement is of the form:

typ v [,*v*]...

where: *typ* is one of INTEGER, REAL, or LOGICAL

v is a variable name, array name, array declarator, function name, or dummy procedure name (18.2.11)

DOUBLE PRECISION and COMPLEX type-statements are not included in the subset.

program in which they appear, but blank common blocks may be of different sizes.

(3) Entities in named common blocks may be initially defined by means of a DATA statement in a block data subprogram, but entities in blank common must not be initially defined (Section 9).

8.3.6 Restrictions on Common and Equivalence. An EQUIVALENCE statement must not cause the storage sequences of two different common blocks in the same program unit to be associated. Equivalence association must not cause a common block storage sequence to be extended by adding storage units preceding the first storage unit of the first entity specified in a COMMON statement for the common block. For example, the following is not permitted:

```
COMMON /X/A
REAL B(2)
EQUIVALENCE (A,B(2))
```

8.4 Type-Statements

A type-statement is used to override or confirm implicit typing and may specify dimension information.

The appearance of the symbolic name of a constant, variable, array, external function, or statement function in a type-statement specifies the data type for that name for all appearances in the program unit. Within a program unit, a name must not have its type explicitly specified more than once.

A type-statement that confirms the type of an intrinsic function whose name appears in the Specific Name column of Table 5 is not required, but is permitted. If a generic function name appears in a type-statement, such an appearance is not sufficient by itself to remove the generic properties from that function.

The name of a main program, subroutine, or block data subprogram must not appear in a type-statement.

8.4.1 INTEGER, REAL, DOUBLE PRECISION, COMPLEX, and LOGICAL Type-Statements. An INTEGER, REAL, DOUBLE PRECISION, COMPLEX, or LOGICAL type-statement is of the form:

<u>typ</u> <u>v</u> [,<u>v</u>]...

where: <u>typ</u> is one of INTEGER, REAL, DOUBLE PRECISION, COMPLEX, or LOGICAL

<u>v</u> is a variable name, array name, array declarator, symbolic name of a constant, function name, or dummy procedure name (18.2.11)

8.4.2 CHARACTER Type-Statement. The form of a CHARACTER type-statement is:

CHARACTER [*len [,]] nam [,nam]...

where: nam is of one of the forms:

v [*len]

a [(d)] [*len]

v is a variable name

a is an array name

a(d) is an array declarator

len is the length (number of characters) of a character variable or character array element, and is called the length specification. len must be an unsigned, nonzero, integer constant.

A length len immediately following the word CHARACTER is the length specification for each entity in the statement not having its own length specification. A length specification immediately following an entity is the length specification for only that entity. Note that for an array the length specified is for each array element. If a length is not specified for an entity, its length is one.

8.4.2 <u>CHARACTER Type-Statement</u>. The form of a CHARACTER type-statement is:

CHARACTER [*<u>len</u> [,]] <u>nam</u> [,<u>nam</u>]...

where: <u>nam</u> is of one of the forms:

<u>v</u> [*<u>len</u>]

<u>a</u> [(<u>d</u>)] [*<u>len</u>]

<u>v</u> is a variable name, symbolic name of a constant, function name, or dummy procedure name

<u>a</u> is an array name

<u>a</u>(<u>d</u>) is an array declarator

<u>len</u> is the length (number of characters) of a character variable, character array element, character constant that has a symbolic name, or character function, and is called the <u>length specification</u>. <u>len</u> is one of the following:

(1) An unsigned, nonzero, integer constant

(2) An integer constant expression (6.1.3.1) enclosed in parentheses and with a positive value

(3) An asterisk in parentheses, (*)

A length <u>len</u> immediately following the word CHARACTER is the length specification for each entity in the statement not having its own length specification. A length specification immediately following an entity is the length specification for only that entity. Note that for an array the length specified is for each array element. If a length is not specified for an entity, its length is one.

An entity declared in a CHARACTER statement must have a length specification that is an integer constant expression, unless that entity is an external function, a dummy argument of an external procedure, or a character constant that has a symbolic name.

If a dummy argument has a <u>len</u> of (*) declared, the dummy argument assumes the length of the associated actual argument for each reference of the subroutine or function. If the associated actual argument is an array name, the length assumed by the dummy argument is the length of an array element in the associated actual argument array.

If an external function has a <u>len</u> of (*) declared in a function subprogram, the function name must appear as the name of a function in a FUNCTION or ENTRY statement in the

8.5 IMPLICIT Statement

An IMPLICIT statement is used to change or confirm the default implied integer and real typing.

The form of an IMPLICIT statement is:

```
IMPLICIT typ (a [,a]...) [,typ (a [,a]...)]...
```

where: *typ* is one of INTEGER, REAL, LOGICAL, or CHARACTER [**len*]

a is either a single letter or a range of single letters in alphabetical order. A range is denoted by the first and last letter of the range separated by a minus. Writing a range of letters $a_1 - a_2$ has the same effect as writing a list of the single letters a_1 through a_2.

len is the length of the character entities and is an unsigned, nonzero, integer constant.

If *len* is not specified, the length is one.

An IMPLICIT statement specifies a type for all variables, arrays, external functions, and statement functions that begin with any letter that appears in the specification, either as a single letter or included in a range of letters. IMPLICIT statements do not change the type of any intrinsic functions. An IMPLICIT statement applies only to the program unit that contains it.

same subprogram. When a reference to such a function is executed, the function assumes the length specified in the referencing program unit.

The length specified for a character function in the program unit that references the function must be an integer constant expression and must agree with the length specified in the subprogram that specifies the function. Note that there always is agreement of length if a *len* of (*) is specified in the subprogram that specifies the function.

If a character constant that has a symbolic name has a *len* of (*) declared, the constant assumes the length of its corresponding constant expression in a PARAMETER statement.

The length specified for a character statement function or statement function dummy argument of type character must be an integer constant expression.

8.5 IMPLICIT Statement

An IMPLICIT statement is used to change or confirm the default implied integer and real typing.

The form of an IMPLICIT statement is:

IMPLICIT *typ* (*a* [,*a*]...) [,*typ* (*a* [,*a*]...)]...

where: *typ* is one of INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, or CHARACTER [**len*]

a is either a single letter or a range of single letters in alphabetical order. A range is denoted by the first and last letter of the range separated by a minus. Writing a range of letters a_1 - a_2 has the same effect as writing a list of the single letters a_1 through a_2.

len is the length of the character entities and is one of the following:

(1) An unsigned, nonzero, integer constant

(2) An integer constant expression (6.1.3.1) enclosed in parentheses and with a positive value

If *len* is not specified, the length is one.

An IMPLICIT statement specifies a type for all variables, arrays, symbolic names of constants, external functions, and statement functions that begin with any letter that appears in the specification, either as a single letter or included in a range of letters. IMPLICIT statements do not change the type of any intrinsic functions. An IMPLICIT statement applies only to the program unit that contains it.

Type specification by an IMPLICIT statement may be overridden or confirmed for any particular variable, array, external function, or statement function name by the appearance of that name in a type-statement. An explicit type specification in a FUNCTION statement overrides an IMPLICIT statement for the name of that function subprogram. Note that the length is also overridden when a particular name appears in a CHARACTER statement.

Within the specification statements of a program unit, IMPLICIT statements must precede all other specification statements. A program unit may contain more than one IMPLICIT statement.

The same letter must not appear as a single letter, or be included in a range of letters, more than once in all of the IMPLICIT statements in a program unit.

8.6 PARAMETER Statement

The PARAMETER statement is not included in the subset.

Type specification by an IMPLICIT statement may be overridden or confirmed for any particular variable, array, symbolic name of a constant, external function, or statement function name by the appearance of that name in a type-statement. An explicit type specification in a FUNCTION statement overrides an IMPLICIT statement for the name of that function subprogram. Note that the length is also overridden when a particular name appears in a CHARACTER or CHARACTER FUNCTION statement.

Within the specification statements of a program unit, IMPLICIT statements must precede all other specification statements except PARAMETER statements. A program unit may contain more than one IMPLICIT statement.

The same letter must not appear as a single letter, or be included in a range of letters, more than once in all of the IMPLICIT statements in a program unit.

8.6 PARAMETER Statement

A PARAMETER statement is used to give a constant a symbolic name.

The form of a PARAMETER statement is:

PARAMETER (p=e [,p=e]...)

where: p is a symbolic name

e is a constant expression (6.7)

If the symbolic name p is of type integer, real, double precision, or complex, the corresponding expression e must be an arithmetic constant expression (6.1.3). If the symbolic name p is of type character or logical, the corresponding expression e must be a character constant expression (6.2.3) or a logical constant expression (6.4.4), respectively.

Each p is the symbolic name of a constant that becomes defined with the value determined from the expression e that appears on the right of the equals, in accordance with the rules for assignment statements (10.1, 10.2, and 10.4).

Any symbolic name of a constant that appears in an expression e must have been defined previously in the same or a different PARAMETER statement in the same program unit.

A symbolic name of a constant must not become defined more than once in a program unit.

If a symbolic name of a constant is not of default implied type, its type must be specified by a type-statement or IMPLICIT statement prior to its first appearance in a PARAMETER statement. If the length specified for the

8.7 EXTERNAL Statement

An EXTERNAL statement is used to identify a symbolic name as representing an external procedure or dummy procedure, and to permit such a name to be used as an actual argument.

The form of an EXTERNAL statement is:

EXTERNAL proc [,proc]...

where each proc is the name of an external procedure or dummy procedure.

Appearance of a name in an EXTERNAL statement declares that name to be an external procedure name or dummy procedure name. If an external procedure name or a dummy procedure name is used as an actual argument in a program unit, it must appear in an EXTERNAL statement in that program unit. Note that a statement function name must not appear in an EXTERNAL statement.

If an intrinsic function name appears in an EXTERNAL statement in a program unit, that name becomes the name of some external procedure and an intrinsic function of the same name is not available for reference in the program unit.

Only one appearance of a symbolic name in all of the EXTERNAL statements of a program unit is permitted.

8.8 INTRINSIC Statement

An INTRINSIC statement is used to identify a symbolic name as representing an intrinsic function (15.3). It also permits a name that represents a specific intrinsic function to be used as an actual argument.

symbolic name of a constant of type character is not the default length of one, its length must be specified in a type-statement or IMPLICIT statement prior to the first appearance of the symbolic name of the constant. Its length must not be changed by subsequent statements including IMPLICIT statements.

Once such a symbolic name is defined, that name may appear in that program unit in any subsequent statement as a primary in an expression or in a DATA statement (9.1). A symbolic name of a constant must not be part of a format specification. A symbolic name of a constant must not be used to form part of another constant, for example, any part of a complex constant.

A symbolic name in a PARAMETER statement may identify only the corresponding constant in that program unit.

8.7 EXTERNAL Statement

An EXTERNAL statement is used to identify a symbolic name as representing an external procedure or dummy procedure, and to permit such a name to be used as an actual argument.

The form of an EXTERNAL statement is:

EXTERNAL _proc_ [,_proc_]...

where each _proc_ is the name of an external procedure, dummy procedure, or block data subprogram.

Appearance of a name in an EXTERNAL statement declares that name to be an external procedure name, dummy procedure name, or block data subprogram name. If an external procedure name or a dummy procedure name is used as an actual argument in a program unit, it must appear in an EXTERNAL statement in that program unit. Note that a statement function name must not appear in an EXTERNAL statement.

If an intrinsic function name appears in an EXTERNAL statement in a program unit, that name becomes the name of some external procedure and an intrinsic function of the same name is not available for reference in the program unit.

Only one appearance of a symbolic name in all of the EXTERNAL statements of a program unit is permitted.

8.8 INTRINSIC Statement

An INTRINSIC statement is used to identify a symbolic name as representing an intrinsic function (15.3). It also permits a name that represents a specific intrinsic function to be used as an actual argument.

The form of an INTRINSIC statement is:

INTRINSIC fun [,fun]...

where each fun is an intrinsic function name.

Appearance of a name in an INTRINSIC statement declares that name to be an intrinsic function name. If a specific name of an intrinsic function is used as an actual argument in a program unit, it must appear in an INTRINSIC statement in that program unit. The names of intrinsic functions for type conversion (INT, IFIX, IDINT, FLOAT, SNGL, REAL, ICHAR), lexical relationship (LGE, LGT, LLE, LLT), and for choosing the largest or smallest value (MAX0, AMAX1, AMAX0, MAX1, MIN0, AMIN1, AMIN0, MIN1) must not be used as actual arguments.

Only one appearance of a symbolic name in all of the INTRINSIC statements of a program unit is permitted. Note that a symbolic name must not appear in both an EXTERNAL and an INTRINSIC statement in a program unit.

8.9 SAVE Statement

A SAVE statement is used to retain the definition status of an entity after the execution of a RETURN or END statement in a subprogram. Within a function or subroutine subprogram, an entity specified by a SAVE statement does not become undefined as a result of the execution of a RETURN or END statement in the subprogram. However, such an entity in a common block may become undefined or redefined in another program unit.

The form of a SAVE statement is:

SAVE a [,a]...

where each a is a named common block name preceded and followed by a slash. Redundant appearances of an item are not permitted.

Dummy argument names, procedure names, variable names, array names, and names of entities in a common block must not appear in a SAVE statement.

The appearance of a common block name preceded and followed by a slash in a SAVE statement has the effect of specifying all of the entities in that common block.

The form of an INTRINSIC statement is:

INTRINSIC <u>fun</u> [,<u>fun</u>]...

where each <u>fun</u> is an intrinsic function name.

Appearance of a name in an INTRINSIC statement declares that name to be an intrinsic function name. If a specific name of an intrinsic function is used as an actual argument in a program unit, it must appear in an INTRINSIC statement in that program unit. The names of intrinsic functions for type conversion (INT, IFIX, IDINT, FLOAT, SNGL, REAL, DBLE, CMPLX, ICHAR, CHAR), lexical relationship (LGE, LGT, LLE, LLT), and for choosing the largest or smallest value (MAX, MAX0, AMAX1, DMAX1, AMAX0, MAX1, MIN, MIN0, AMIN1, DMIN1, AMIN0, MIN1) must not be used as actual arguments.

The appearance of a generic function name in an INTRINSIC statement does not cause that name to lose its generic property.

Only one appearance of a symbolic name in all of the INTRINSIC statements of a program unit is permitted. Note that a symbolic name must not appear in both an EXTERNAL and an INTRINSIC statement in a program unit.

8.9 <u>SAVE Statement</u>

A SAVE statement is used to retain the definition status of an entity after the execution of a RETURN or END statement in a subprogram. Within a function or subroutine subprogram, an entity specified by a SAVE statement does not become undefined as a result of the execution of a RETURN or END statement in the subprogram. However, such an entity in a common block may become undefined or redefined in another program unit.

The form of a SAVE statement is:

SAVE [<u>a</u> [,<u>a</u>]...]

where each <u>a</u> is a named common block name preceded and followed by a slash, a variable name, or an array name. Redundant appearances of an item are not permitted.

Dummy argument names, procedure names, and names of entities in a common block must not appear in a SAVE statement.

A SAVE statement without a list is treated as though it contained the names of all allowable items in that program unit.

The appearance of a common block name preceded and followed by a slash in a SAVE statement has the effect of specifying all of the entities in that common block.

If a particular common block name is specified by a SAVE statement in a subprogram of an executable program, it must be specified by a SAVE statement in every subprogram in which that common block appears.

A SAVE statement is optional in a main program and has no effect.

If a named common block is specified in a SAVE statement in a subprogram, the current values of the entities in the common block storage sequence (8.3.3) at the time a RETURN or END statement is executed are made available to the next program unit that specifies that common block name in the execution sequence of an executable program.

If a named common block is specified in the main program unit, the current values of the common block storage sequence are made available to each subprogram that specifies that named common block; a SAVE statement in the subprogram has no effect.

The definition status of each entity in the named common block storage sequence depends on the association that has been established for the common block storage sequence (17.2 and 17.3).

The execution of a RETURN statement or an END statement within a subprogram causes all entities within the subprogram to become undefined except for the following:

(1) Entities specified by SAVE statements

(2) Entities in blank common

(3) Initially defined entities that have neither been redefined nor become undefined

(4) Entities in a named common block that appears in the subprogram and appears in at least one other program unit that is referencing, either directly or indirectly, that subprogram

If a particular common block name is specified by a SAVE statement in a subprogram of an executable program, it must be specified by a SAVE statement in every subprogram in which that common block appears.

A SAVE statement is optional in a main program and has no effect.

If a named common block is specified in a SAVE statement in a subprogram, the current values of the entities in the common block storage sequence (8.3.3) at the time a RETURN or END statement is executed are made available to the next program unit that specifies that common block name in the execution sequence of an executable program.

If a named common block is specified in the main program unit, the current values of the common block storage sequence are made available to each subprogram that specifies that named common block; a SAVE statement in the subprogram has no effect.

The definition status of each entity in the named common block storage sequence depends on the association that has been established for the common block storage sequence (17.2 and 17.3).

If a local entity that is specified by a SAVE statement and is not in a common block is in a defined state at the time a RETURN or END statement is executed in a subprogram, that entity is defined with the same value at the next reference of that subprogram.

The execution of a RETURN statement or an END statement within a subprogram causes all entities within the subprogram to become undefined except for the following:

(1) Entities specified by SAVE statements

(2) Entities in blank common

(3) Initially defined entities that have neither been redefined nor become undefined

(4) Entities in a named common block that appears in the subprogram and appears in at least one other program unit that is referencing, either directly or indirectly, that subprogram

9. DATA STATEMENT

A DATA statement is used to provide initial values for variables, arrays, and array elements. A DATA statement is nonexecutable and may appear in a program unit after the specification statements and before any statement function statements or executable statements.

All initially defined entities are defined when an executable program begins execution. All entities not initially defined, or associated with an initially defined entity, are undefined at the beginning of execution of an executable program.

9.1 Form of a DATA Statement

The form of a DATA statement is:

DATA <u>nlist</u> /<u>clist</u>/ [[,] <u>nlist</u> /<u>clist</u>/]...

where: <u>nlist</u> is a list (2.10) of variable names, array names, and array element names

<u>clist</u> is a list of the form:

<u>a</u> [,<u>a</u>]...

where <u>a</u> is one of the forms:

<u>c</u>
<u>r</u>*<u>c</u>

<u>c</u> is a constant

<u>r</u> is a nonzero, unsigned, integer constant. The <u>r</u>*<u>c</u> form is equivalent to <u>r</u> successive appearances of the constant <u>c</u>.

9.2 DATA Statement Restrictions

Names of dummy arguments, functions, and entities in common (including entities associated with an entity in common) must not appear in the list <u>nlist</u>.

There must be the same number of items specified by each list <u>nlist</u> and its corresponding list <u>clist</u>. There is a one-to-one correspondence between the items specified by <u>nlist</u> and the constants specified by <u>clist</u> such that the first item of <u>nlist</u> corresponds to the first constant of <u>clist</u>, etc. By this correspondence, the initial value is established and the entity is initially defined. If an

9. DATA STATEMENT

A DATA statement is used to provide initial values for variables, arrays, array elements, and substrings. A DATA statement is nonexecutable and may appear in a program unit anywhere after the specification statements, if any.

All initially defined entities are defined when an executable program begins execution. All entities not initially defined, or associated with an initially defined entity, are undefined at the beginning of execution of an executable program.

9.1 Form of a DATA Statement

The form of a DATA statement is:

DATA _nlist_ /_clist_/ [[,] _nlist_ /_clist_/]...

where: _nlist_ is a list (2.10) of variable names, array names, array element names, substring names, and implied-DO lists

clist is a list of the form:

_a_ [,_a_]...

where _a_ is one of the forms:

_c_
_r_*_c_

_c_ is a constant or the symbolic name of a constant

_r_ is a nonzero, unsigned, integer constant or the symbolic name of such a constant. The _r_*_c_ form is equivalent to _r_ successive appearances of the constant _c_.

9.2 DATA Statement Restrictions

Names of dummy arguments, functions, and entities in blank common (including entities associated with an entity in blank common) must not appear in the list _nlist_. Names of entities in a named common block may appear in the list _nlist_ only within a block data subprogram.

There must be the same number of items specified by each list _nlist_ and its corresponding list _clist_. There is a one-to-one correspondence between the items specified by _nlist_ and the constants specified by _clist_ such that the first item of _nlist_ corresponds to the first constant of _clist_, etc. By this correspondence, the initial value is established and the entity is initially defined. If an array name without a subscript is in the list, there must be one constant for each element of that array. The ordering of array elements is determined by the array element subscript value (5.2.4).

The type of the _nlist_ entity and the type of the corresponding _clist_ constant must agree.

Any variable or array element may be initially defined except for:

(1) an entity that is a dummy argument,

(2) an entity in common, which includes an entity associated with an entity in common, or

(3) a variable in a function subprogram whose name is also the name of the function subprogram.

A variable or array element must not be initially defined more than once in an executable program. If two entities are associated, only one may be initially defined in a DATA statement in the same executable program.

Each subscript expression in the list _nlist_ must be an integer constant.

9.3 Implied-DO in a DATA Statement

Implied-DO lists in DATA statements are not included in the subset.

array name without a subscript is in the list, there must be one constant for each element of that array. The ordering of array elements is determined by the array element subscript value (5.2.4).

The type of the _nlist_ entity and the type of the corresponding _clist_ constant must agree when either is of type character or logical. When the _nlist_ entity is of type integer, real, double precision, or complex, the corresponding _clist_ constant must also be of type integer, real, double precision, or complex; if necessary, the _clist_ constant is converted to the type of the _nlist_ entity according to the rules for arithmetic conversion (Table 4). Note that if an _nlist_ entity is of type double precision and the _clist_ constant is of type real, the processor may supply more precision derived from the constant than can be contained in a real datum.

Any variable, array element, or substring may be initially defined except for:

(1) an entity that is a dummy argument,

(2) an entity in blank common, which includes an entity associated with an entity in blank common, or

(3) a variable in a function subprogram whose name is also the name of the function subprogram or an entry in the function subprogram.

A variable, array element, or substring must not be initially defined more than once in an executable program. If two entities are associated, only one may be initially defined in a DATA statement in the same executable program.

Each subscript expression in the list _nlist_ must be an integer constant expression except for implied-DO-variables as noted in 9.3. Each substring expression in the list _nlist_ must be an integer constant expression.

9.3 _Implied-DO in a DATA Statement_

The form of an implied-DO list in a DATA statement is:

(_dlist_, _i_ = $\underline{m}_1$, $\underline{m}_2$ [, $\underline{m}_3$])

where: _dlist_ is a list of array element names and implied-DO lists

_i_ is the name of an integer variable, called the _implied-DO-variable_

$\underline{m}_1$, $\underline{m}_2$, $\underline{m}_3$ are each an integer constant expression, except that the expression may contain implied-DO-variables of other implied-DO lists that have this implied-DO list within their ranges

9.4 Character Constant in a DATA Statement

An entity in the list _nlist_ that corresponds to a character constant must be of type character.

If the length of the character entity in the list _nlist_ is greater than the length of its corresponding character constant, the additional rightmost characters in the entity are initially defined with blank characters.

The length of the character entity in the list _nlist_ must be greater than or equal to the length of its corresponding character constant.

Note that initial definition of a character entity causes definition of all of the characters in the entity, and that each character constant initially defines exactly one variable or array element.

The range of an implied-DO list is the list dlist. An iteration count and the values of the implied-DO-variable are established from m_1, m_2, and m_3 exactly as for a DO-loop (11.10), except that the iteration count must be positive. When an implied-DO list appears in a DATA statement, the list items in dlist are specified once for each iteration of the implied-DO list with the appropriate substitution of values for any occurrence of the implied-DO-variable i. The appearance of an implied-DO-variable name in a DATA statement does not affect the definition status of a variable of the same name in the same program unit.

Each subscript expression in the list dlist must be an integer constant expression, except that the expression may contain implied-DO-variables of implied-DO lists that have the subscript expression within their ranges.

The following is an example of a DATA statement that contains implied-DO lists:

```
DATA (( X(J,I), I=1,J), J=1,5) / 15*0. /
```

9.4 Character Constant in a DATA Statement

An entity in the list nlist that corresponds to a character constant must be of type character.

If the length of the character entity in the list nlist is greater than the length of its corresponding character constant, the additional rightmost characters in the entity are initially defined with blank characters.

If the length of the character entity in the list nlist is less than the length of its corresponding character constant, the additional rightmost characters in the constant are ignored.

Note that initial definition of a character entity causes definition of all of the characters in the entity, and that each character constant initially defines exactly one variable, array element, or substring.

10. ASSIGNMENT STATEMENTS

Completion of execution of an assignment statement causes definition of an entity.

There are four kinds of assignment statements:

(1) Arithmetic

(2) Logical

(3) Statement label (ASSIGN)

(4) Character

10.1 Arithmetic Assignment Statement

The form of an arithmetic assignment statement is:

$\underline{v} = \underline{e}$

where: $\underline{v}$ is the name of a variable or array element of type integer or real

$\underline{e}$ is an arithmetic expression

Execution of an arithmetic assignment statement causes the evaluation of the expression $\underline{e}$ by the rules in Section 6, conversion of $\underline{e}$ to the type of $\underline{v}$, and definition and assignment of $\underline{v}$ with the resulting value, as established by the rules in Table 4.

Table 4

Arithmetic Conversion and Assignment of $\underline{e}$ to $\underline{v}$

Type of $\underline{v}$	Type of $\underline{e}$	Value Assigned
Integer	Integer	$\underline{e}$
Real	Real	$\underline{e}$
Integer	Real	INT($\underline{e}$)
Real	Integer	REAL($\underline{e}$)

The INT and REAL functions in the "Value Assigned" column of Table 4 are intrinsic functions described in the "Specific Name" column of Table 5 (15.10).

10. ASSIGNMENT STATEMENTS

Completion of execution of an assignment statement causes definition of an entity.

There are four kinds of assignment statements:

(1) Arithmetic

(2) Logical

(3) Statement label (ASSIGN)

(4) Character

10.1 Arithmetic Assignment Statement

The form of an arithmetic assignment statement is:

$\underline{v} = \underline{e}$

where: $\underline{v}$ is the name of a variable or array element of type integer, real, double precision, or complex

$\underline{e}$ is an arithmetic expression

Execution of an arithmetic assignment statement causes the evaluation of the expression $\underline{e}$ by the rules in Section 6, conversion of $\underline{e}$ to the type of $\underline{v}$, and definition and assignment of $\underline{v}$ with the resulting value, as established by the rules in Table 4.

Table 4

Arithmetic Conversion and Assignment of $\underline{e}$ to $\underline{v}$

Type of $\underline{v}$	Value Assigned
Integer	INT($\underline{e}$)
Real	REAL($\underline{e}$)
Double Precision	DBLE($\underline{e}$)
Complex	CMPLX($\underline{e}$)

The functions in the "Value Assigned" column of Table 4 are generic functions described in Table 5 (15.10).

10.2 Logical Assignment Statement

The form of a logical assignment statement is:

v = *e*

where: *v* is the name of a logical variable or logical array element

e is a logical expression

Execution of a logical assignment statement causes the evaluation of the logical expression *e* and the assignment and definition of *v* with the value of *e*. Note that *e* must have a value of either true or false.

10.3 Statement Label Assignment (ASSIGN) Statement

The form of a statement label assignment statement is:

ASSIGN *s* TO *i*

where: *s* is a statement label

i is an integer variable name

Execution of an ASSIGN statement causes the statement label *s* to be assigned to the integer variable *i*. The statement label must be the label of a statement that appears in the same program unit as the ASSIGN statement. The statement label must be the label of an executable statement or a FORMAT statement.

Execution of a statement label assignment statement is the only way that a variable may be defined with a statement label value.

A variable must be defined with a statement label value when referenced in an assigned GO TO statement (11.3) or as a format identifier (12.4) in an input/output statement. While defined with a statement label value, the variable must not be referenced in any other way.

An integer variable defined with a statement label value may be redefined with the same or a different statement label value or an integer value.

10.4 Character Assignment Statement

The form of a character assignment statement is:

v = *e*

where: *v* is the name of a character variable or character array element

10.2 Logical Assignment Statement

The form of a logical assignment statement is:

v = *e*

where: *v* is the name of a logical variable or logical array element

e is a logical expression

Execution of a logical assignment statement causes the evaluation of the logical expression *e* and the assignment and definition of *v* with the value of *e*. Note that *e* must have a value of either true or false.

10.3 Statement Label Assignment (ASSIGN) Statement

The form of a statement label assignment statement is:

ASSIGN *s* TO *i*

where: *s* is a statement label

i is an integer variable name

Execution of an ASSIGN statement causes the statement label *s* to be assigned to the integer variable *i*. The statement label must be the label of a statement that appears in the same program unit as the ASSIGN statement. The statement label must be the label of an executable statement or a FORMAT statement.

Execution of a statement label assignment statement is the only way that a variable may be defined with a statement label value.

A variable must be defined with a statement label value when referenced in an assigned GO TO statement (11.3) or as a format identifier (12.4) in an input/output statement. While defined with a statement label value, the variable must not be referenced in any other way.

An integer variable defined with a statement label value may be redefined with the same or a different statement label value or an integer value.

10.4 Character Assignment Statement

The form of a character assignment statement is:

v = *e*

where: *v* is the name of a character variable, character array element, or character substring *e* is a character expression

Execution of a character assignment statement causes the evaluation of the expression *e* and the assignment and definition of *v* with the value of *e*. None of the character positions being defined in *v* may be referenced in *e*. *v* and *e* may have different lengths. If the length of *v* is greater than the length of *e*, the effect is as though *e* were extended to the right with blank characters until it is the same length as *v* and then assigned. If the length of *v* is less than the length of *e*, the effect is as though *e* were truncated from the right until it is the same length as *v* and then assigned.

e is a character expression

Execution of a character assignment statement causes the evaluation of the expression *e* and the assignment and definition of *v* with the value of *e*. None of the character positions being defined in *v* may be referenced in *e*. *v* and *e* may have different lengths. If the length of *v* is greater than the length of *e*, the effect is as though *e* were extended to the right with blank characters until it is the same length as *v* and then assigned. If the length of *v* is less than the length of *e*, the effect is as though *e* were truncated from the right until it is the same length as *v* and then assigned.

Only as much of the value of *e* must be defined as is needed to define *v*. In the example:

```
CHARACTER A*2, B*4
A=B
```

the assignment A=B requires that the substring B(1:2) be defined. It does not require that the substring B(3:4) be defined.

If *v* is a substring, *e* is assigned only to the substring. The definition status of substrings not specified by *v* is unchanged.

11. CONTROL STATEMENTS

Control statements may be used to control the execution sequence.

There are sixteen control statements:

(1) Unconditional GO TO

(2) Computed GO TO

(3) Assigned GO TO

(4) Arithmetic IF

(5) Logical IF

(6) Block IF

(7) ELSE IF

(8) ELSE

(9) END IF

(10) DO

(11) CONTINUE

(12) STOP

(13) PAUSE

(14) END

(15) CALL

(16) RETURN

The CALL and RETURN statements are described in Section 15.

11.1 Unconditional GO TO Statement

The form of an unconditional GO TO statement is:

GO TO *s*

where *s* is the statement label of an executable statement that appears in the same program unit as the unconditional GO TO statement.

Execution of an unconditional GO TO statement causes a transfer of control so that the statement identified by the statement label is executed next.

11. CONTROL STATEMENTS

Control statements may be used to control the execution sequence.

There are sixteen control statements:

(1) Unconditional GO TO

(2) Computed GO TO

(3) Assigned GO TO

(4) Arithmetic IF

(5) Logical IF

(6) Block IF

(7) ELSE IF

(8) ELSE

(9) END IF

(10) DO

(11) CONTINUE

(12) STOP

(13) PAUSE

(14) END

(15) CALL

(16) RETURN

The CALL and RETURN statements are described in Section 15.

11.1 Unconditional GO TO Statement

The form of an unconditional GO TO statement is:

GO TO *s*

where *s* is the statement label of an executable statement that appears in the same program unit as the unconditional GO TO statement.

Execution of an unconditional GO TO statement causes a transfer of control so that the statement identified by the statement label is executed next.

11.2 Computed GO TO Statement

The form of a computed GO TO statement is:

GO TO (s [,s]...) [,] i

where: i is an integer variable name

s is the statement label of an executable statement that appears in the same program unit as the computed GO TO statement. The same statement label may appear more than once in the same computed GO TO statement.

Execution of a computed GO TO statement causes a transfer of control so that the statement identified by the ith statement label in the list of statement labels is executed next, provided that $1 \le i \le n$, where n is the number of statement labels in the list of statement labels. If $i<1$ or $i>n$, the execution sequence continues as though a CONTINUE statement were executed.

11.3 Assigned GO TO Statement

The form of an assigned GO TO statement is:

GO TO i [[,] (s [,s]...)]

where: i is an integer variable name

s is the statement label of an executable statement that appears in the same program unit as the assigned GO TO statement. The same statement label may appear more than once in the same assigned GO TO statement.

At the time of execution of an assigned GO TO statement, the variable i must be defined with the value of a statement label of an executable statement that appears in the same program unit. Note that the variable may be defined with a statement label value only by an ASSIGN statement (10.3) in the same program unit as the assigned GO TO statement. The execution of the assigned GO TO statement causes a transfer of control so that the statement identified by that statement label is executed next.

If the parenthesized list is present, the statement label assigned to i must be one of the statement labels in the list.

11.4 Arithmetic IF Statement

The form of an arithmetic IF statement is:

IF (e) s_1 , s_2 , s_3

11.2 Computed GO TO Statement

The form of a computed GO TO statement is:

GO TO (*s* [,*s*]...) [,] *i*

where: *i* is an integer expression

s is the statement label of an executable statement that appears in the same program unit as the computed GO TO statement. The same statement label may appear more than once in the same computed GO TO statement.

Execution of a computed GO TO statement causes evaluation of the expression *i*. The evaluation of *i* is followed by a transfer of control so that the statement identified by the *i*th statement label in the list of statement labels is executed next, provided that $1 \le i \le n$, where *n* is the number of statement labels in the list of statement labels. If $i<1$ or $i>n$, the execution sequence continues as though a CONTINUE statement were executed.

11.3 Assigned GO TO Statement

The form of an assigned GO TO statement is:

GO TO *i* [[,] (*s* [,*s*]...)]

where: *i* is an integer variable name

s is the statement label of an executable statement that appears in the same program unit as the assigned GO TO statement. The same statement label may appear more than once in the same assigned GO TO statement.

At the time of execution of an assigned GO TO statement, the variable *i* must be defined with the value of a statement label of an executable statement that appears in the same program unit. Note that the variable may be defined with a statement label value only by an ASSIGN statement (10.3) in the same program unit as the assigned GO TO statement. The execution of the assigned GO TO statement causes a transfer of control so that the statement identified by that statement label is executed next.

If the parenthesized list is present, the statement label assigned to *i* must be one of the statement labels in the list.

11.4 Arithmetic IF Statement

The form of an arithmetic IF statement is:

IF (*e*) s_1 , s_2 , s_3

where: *e* is an integer or real expression

s_1, s_2, and s_3 are each the statement label of an executable statement that appears in the same program unit as the arithmetic IF statement. The same statement label may appear more than once in the same arithmetic IF statement.

Execution of an arithmetic IF statement causes evaluation of the expression *e* followed by a transfer of control. The statement identified by s_1, s_2, or s_3 is executed next as the value of *e* is less than zero, equal to zero, or greater than zero, respectively.

11.5 Logical IF Statement

The form of a logical IF statement is:

IF (*e*) *st*

where: *e* is a logical expression

st is any executable statement except a DO, block IF, ELSE IF, ELSE, END IF, END, or another logical IF statement

Execution of a logical IF statement causes evaluation of the expression *e*. If the value of *e* is true, statement *st* is executed. If the value of *e* is false, statement *st* is not executed and the execution sequence continues as though a CONTINUE statement were executed.

Note that the execution of a function reference in the expression *e* of a logical IF statement is permitted to affect entities in the statement *st*.

11.6 Block IF Statement

The block IF statement is used with the END IF statement and, optionally, the ELSE IF and ELSE statements to control the execution sequence.

The form of a block IF statement is:

IF (*e*) THEN

where *e* is a logical expression.

11.6.1 IF-Level. The IF-level of a statement *s* is

$n_1 - n_2$

where n_1 is the number of block IF statements from the beginning of the program unit up to and including *s*, and n_2

where: e is an integer, real, or double precision expression

s_1, s_2, and s_3 are each the statement label of an executable statement that appears in the same program unit as the arithmetic IF statement. The same statement label may appear more than once in the same arithmetic IF statement.

Execution of an arithmetic IF statement causes evaluation of the expression e followed by a transfer of control. The statement identified by s_1, s_2, or s_3 is executed next as the value of e is less than zero, equal to zero, or greater than zero, respectively.

11.5 Logical IF Statement

The form of a logical IF statement is:

IF (e) st

where: e is a logical expression

st is any executable statement except a DO, block IF, ELSE IF, ELSE, END IF, END, or another logical IF statement

Execution of a logical IF statement causes evaluation of the expression e. If the value of e is true, statement st is executed. If the value of e is false, statement st is not executed and the execution sequence continues as though a CONTINUE statement were executed.

Note that the execution of a function reference in the expression e of a logical IF statement is permitted to affect entities in the statement st.

11.6 Block IF Statement

The block IF statement is used with the END IF statement and, optionally, the ELSE IF and ELSE statements to control the execution sequence.

The form of a block IF statement is:

IF (e) THEN

where e is a logical expression.

11.6.1 IF-Level.

The *IF-level* of a statement s is

$n_1 - n_2$

where n_1 is the number of block IF statements from the beginning of the program unit up to and including s, and n_2

is the number of END IF statements in the program unit up to but not including s.

The IF-level of every statement must be zero or positive. The IF-level of each block IF, ELSE IF, ELSE, and END IF statement must be positive. The IF-level of the END statement of each program unit must be zero.

11.6.2 IF-Block. An IF-block consists of all of the executable statements that appear following the block IF statement up to, but not including, the next ELSE IF, ELSE, or END IF statement that has the same IF-level as the block IF statement. An IF-block may be empty.

11.6.3 Execution of a Block IF Statement. Execution of a block IF statement causes evaluation of the expression e. If the value of e is true, normal execution sequence continues with the first statement of the IF-block. If the value of e is true and the IF-block is empty, control is transferred to the next END IF statement that has the same IF-level as the block IF statement. If the value of e is false, control is transferred to the next ELSE IF, ELSE, or END IF statement that has the same IF-level as the block IF statement.

Transfer of control into an IF-block from outside the IF-block is prohibited.

If the execution of the last statement in the IF-block does not result in a transfer of control, control is transferred to the next END IF statement that has the same IF-level as the block IF statement that precedes the IF-block.

11.7 ELSE IF Statement

The form of an ELSE IF statement is:

ELSE IF (e) THEN

where e is a logical expression.

11.7.1 ELSE IF-Block. An ELSE IF-block consists of all of the executable statements that appear following the ELSE IF statement up to, but not including, the next ELSE IF, ELSE, or END IF statement that has the same IF-level as the ELSE IF statement. An ELSE IF-block may be empty.

11.7.2 Execution of an ELSE IF Statement. Execution of an ELSE IF statement causes evaluation of the expression e. If the value of e is true, normal execution sequence continues with the first statement of the ELSE IF-block. If the value of e is true and the ELSE IF-block is empty, control is transferred to the next END IF statement that has the same IF-level as the ELSE IF statement. If the value of e is false, control is transferred to the next ELSE IF, ELSE, or is the number of END IF statements in the program unit up to but not including s.

The IF-level of every statement must be zero or positive. The IF-level of each block IF, ELSE IF, ELSE, and END IF statement must be positive. The IF-level of the END statement of each program unit must be zero.

11.6.2 IF-Block. An IF-block consists of all of the executable statements that appear following the block IF statement up to, but not including, the next ELSE IF, ELSE, or END IF statement that has the same IF-level as the block IF statement. An IF-block may be empty.

11.6.3 Execution of a Block IF Statement. Execution of a block IF statement causes evaluation of the expression e. If the value of e is true, normal execution sequence continues with the first statement of the IF-block. If the value of e is true and the IF-block is empty, control is transferred to the next END IF statement that has the same IF-level as the block IF statement. If the value of e is false, control is transferred to the next ELSE IF, ELSE, or END IF statement that has the same IF-level as the block IF statement.

Transfer of control into an IF-block from outside the IF-block is prohibited.

If the execution of the last statement in the IF-block does not result in a transfer of control, control is transferred to the next END IF statement that has the same IF-level as the block IF statement that precedes the IF-block.

11.7 ELSE IF Statement

The form of an ELSE IF statement is:

```
ELSE IF (e) THEN
```

where e is a logical expression.

11.7.1 ELSE IF-Block. An ELSE IF-block consists of all of the executable statements that appear following the ELSE IF statement up to, but not including, the next ELSE IF, ELSE, or END IF statement that has the same IF-level as the ELSE IF statement. An ELSE IF-block may be empty.

11.7.2 Execution of an ELSE IF Statement. Execution of an ELSE IF statement causes evaluation of the expression e. If the value of e is true, normal execution sequence continues with the first statement of the ELSE IF-block. If the value of e is true and the ELSE IF-block is empty, control is transferred to the next END IF statement that has the same IF-level as the ELSE IF statement. If the value of e is false, control is transferred to the next ELSE IF, ELSE, or

END IF statement that has the same IF-level as the ELSE IF statement.

Transfer of control into an ELSE IF-block from outside the ELSE IF-block is prohibited. The statement label, if any, of the ELSE IF statement must not be referenced by any statement.

If execution of the last statement in the ELSE IF-block does not result in a transfer of control, control is transferred to the next END IF statement that has the same IF-level as the ELSE IF statement that precedes the ELSE IF-block.

11.8 ELSE Statement

The form of an ELSE statement is:

```
ELSE
```

11.8.1 ELSE-Block. An ELSE-block consists of all of the executable statements that appear following the ELSE statement up to, but not including, the next END IF statement that has the same IF-level as the ELSE statement. An ELSE-block may be empty.

An END IF statement of the same IF-level as the ELSE statement must appear before the appearance of an ELSE IF or ELSE statement of the same IF-level.

11.8.2 Execution of an ELSE Statement. Execution of an ELSE statement has no effect.

Transfer of control into an ELSE-block from outside the ELSE-block is prohibited. The statement label, if any, of an ELSE statement must not be referenced by any statement.

11.9 END IF Statement

The form of an END IF statement is:

```
END IF
```

Execution of an END IF statement has no effect.

For each block IF statement there must be a corresponding END IF statement in the same program unit. A corresponding END IF statement is the next END IF statement that has the same IF-level as the block IF statement.

11.10 DO Statement

A DO statement is used to specify a loop, called a DO-loop.

END IF statement that has the same IF-level as the ELSE IF statement.

Transfer of control into an ELSE IF-block from outside the ELSE IF-block is prohibited. The statement label, if any, of the ELSE IF statement must not be referenced by any statement.

If execution of the last statement in the ELSE IF-block does not result in a transfer of control, control is transferred to the next END IF statement that has the same IF-level as the ELSE IF statement that precedes the ELSE IF-block.

11.8 ELSE Statement

The form of an ELSE statement is:

    ELSE

11.8.1 ELSE-Block. An *ELSE-block* consists of all of the executable statements that appear following the ELSE statement up to, but not including, the next END IF statement that has the same IF-level as the ELSE statement. An ELSE-block may be empty.

An END IF statement of the same IF-level as the ELSE statement must appear before the appearance of an ELSE IF or ELSE statement of the same IF-level.

11.8.2 Execution of an ELSE Statement. Execution of an ELSE statement has no effect.

Transfer of control into an ELSE-block from outside the ELSE-block is prohibited. The statement label, if any, of an ELSE statement must not be referenced by any statement.

11.9 END IF Statement

The form of an END IF statement is:

    END IF

Execution of an END IF statement has no effect.

For each block IF statement there must be a corresponding END IF statement in the same program unit. A *corresponding END IF statement* is the next END IF statement that has the same IF-level as the block IF statement.

11.10 DO Statement

A DO statement is used to specify a loop, called a *DO-loop*.

The form of a DO statement is:

DO s [,] i = e_1, e_2 [,e_3]

where: s is the statement label of an executable statement. The statement identified by s, called the terminal statement of the DO-loop, must follow the DO statement in the sequence of statements within the same program unit as the DO statement.

i is the name of an integer variable, called the DO-variable

e_1, e_2, and e_3 are each an integer constant or integer variable name

The terminal statement of a DO-loop must not be an unconditional GO TO, assigned GO TO, arithmetic IF, block IF, ELSE IF, ELSE, END IF, RETURN, STOP, END, or DO statement. If the terminal statement of a DO-loop is a logical IF statement, it may contain any executable statement except a DO, block IF, ELSE IF, ELSE, END IF, END, or another logical IF statement.

11.10.1 Range of a DO-Loop. The range of a DO-loop consists of all of the executable statements that appear following the DO statement that specifies the DO-loop, up to and including the terminal statement of the DO-loop.

If a DO statement appears within the range of a DO-loop, the range of the DO-loop specified by that DO statement must be contained entirely within the range of the outer DO-loop. More than one DO-loop may have the same terminal statement.

If a DO statement appears within an IF-block, ELSE IF-block, or ELSE-block, the range of that DO-loop must be contained entirely within that IF-block, ELSE IF-block, or ELSE-block, respectively.

If a block IF statement appears within the range of a DO-loop, the corresponding END IF statement must also appear within the range of that DO-loop.

11.10.2 Active and Inactive DO-Loops. A DO-loop is either active or inactive. Initially inactive, a DO-loop becomes active only when its DO statement is executed.

Once active, the DO-loop becomes inactive only when:

(1) its iteration count is tested (11.10.4) and determined to be zero,

(2) a RETURN statement is executed within its range,

The form of a DO statement is:

DO $\underline{s}$ [,] $\underline{i}$ = $\underline{e}_1$, $\underline{e}_2$ [,$\underline{e}_3$]

where: $\underline{s}$ is the statement label of an executable statement. The statement identified by $\underline{s}$, called the terminal statement of the DO-loop, must follow the DO statement in the sequence of statements within the same program unit as the DO statement.

$\underline{i}$ is the name of an integer, real, or double precision variable, called the DO-variable

$\underline{e}_1$, $\underline{e}_2$, and $\underline{e}_3$ are each an integer, real, or double precision expression

The terminal statement of a DO-loop must not be an unconditional GO TO, assigned GO TO, arithmetic IF, block IF, ELSE IF, ELSE, END IF, RETURN, STOP, END, or DO statement. If the terminal statement of a DO-loop is a logical IF statement, it may contain any executable statement except a DO, block IF, ELSE IF, ELSE, END IF, END, or another logical IF statement.

11.10.1 Range of a DO-Loop. The range of a DO-loop consists of all of the executable statements that appear following the DO statement that specifies the DO-loop, up to and including the terminal statement of the DO-loop.

If a DO statement appears within the range of a DO-loop, the range of the DO-loop specified by that DO statement must be contained entirely within the range of the outer DO-loop. More than one DO-loop may have the same terminal statement.

If a DO statement appears within an IF-block, ELSE IF-block, or ELSE-block, the range of that DO-loop must be contained entirely within that IF-block, ELSE IF-block, or ELSE-block, respectively.

If a block IF statement appears within the range of a DO-loop, the corresponding END IF statement must also appear within the range of that DO-loop.

11.10.2 Active and Inactive DO-Loops. A DO-loop is either active or inactive. Initially inactive, a DO-loop becomes active only when its DO statement is executed.

Once active, the DO-loop becomes inactive only when:

(1) its iteration count is tested (11.10.4) and determined to be zero,

(2) a RETURN statement is executed within its range,

(3) control is transferred to a statement that is in the same program unit and is outside the range of the DO-loop, or

(4) any STOP statement in the executable program is executed, or execution is terminated for any other reason (12.6).

Execution of a function reference or CALL statement that appears in the range of a DO-loop does not cause the DO-loop to become inactive.

When a DO-loop becomes inactive, the DO-variable of the DO-loop retains its last defined value.

11.10.3 Executing a DO Statement. The effect of executing a DO statement is to perform the following steps in sequence:

(1) The initial parameter $\underline{m}_1$, the terminal parameter $\underline{m}_2$, and the incrementation parameter $\underline{m}_3$ are established from $\underline{e}_1$, $\underline{e}_2$, and $\underline{e}_3$, respectively. If $\underline{e}_3$ does not appear, $\underline{m}_3$ has a value of one. $\underline{m}_3$ must not have a value of zero.

(2) The DO-variable becomes defined with the value of the initial parameter $\underline{m}_1$.

(3) The iteration count is established and is the value of the expression

$$\text{MAX0}(\ (\ (\underline{m}_2 - \underline{m}_1 + \underline{m}_3)/\underline{m}_3),\ 0)$$

Note that the iteration count is zero whenever:

$\underline{m}_1 > \underline{m}_2$ and $\underline{m}_3 > 0$, or

$\underline{m}_1 < \underline{m}_2$ and $\underline{m}_3 < 0$.

At the completion of execution of the DO statement, loop control processing begins.

11.10.4 Loop Control Processing. Loop control processing determines if further execution of the range of the DO-loop is required. The iteration count is tested. If it is not zero, execution of the first statement in the range of the DO-loop begins. If the iteration count is zero, the DO-loop becomes inactive. If, as a result, all of the DO-loops sharing the terminal statement of this DO-loop are inactive, normal execution continues with execution of the next executable statement following the terminal statement. However, if some of the DO-loops sharing the terminal

(3) control is transferred to a statement that is in the same program unit and is outside the range of the DO-loop, or

(4) any STOP statement in the executable program is executed, or execution is terminated for any other reason (12.6).

Execution of a function reference or CALL statement that appears in the range of a DO-loop does not cause the DO-loop to become inactive, except when control is returned by means of an alternate return specifier in a CALL statement to a statement that is not in the range of the DO-loop.

When a DO-loop becomes inactive, the DO-variable of the DO-loop retains its last defined value.

11.10.3 <u>Executing a DO Statement</u>. The effect of executing a DO statement is to perform the following steps in sequence:

(1) The <u>initial parameter</u> $\underline{m}_1$, the <u>terminal parameter</u> $\underline{m}_2$, and the <u>incrementation parameter</u> $\underline{m}_3$ are established by evaluating $\underline{e}_1$, $\underline{e}_2$, and $\underline{e}_3$, respectively, including, if necessary, conversion to the type of the DO-variable according to the rules for arithmetic conversion (Table 4). If $\underline{e}_3$ does not appear, $\underline{m}_3$ has a value of one. $\underline{m}_3$ must not have a value of zero.

(2) The DO-variable becomes defined with the value of the initial parameter $\underline{m}_1$.

(3) The iteration count is established and is the value of the expression

MAX(INT(($\underline{m}_2$ - $\underline{m}_1$ + $\underline{m}_3$)/$\underline{m}_3$), 0)

Note that the iteration count is zero whenever:

$\underline{m}_1 > \underline{m}_2$ and $\underline{m}_3 > 0$, or

$\underline{m}_1 < \underline{m}_2$ and $\underline{m}_3 < 0$.

At the completion of execution of the DO statement, loop control processing begins.

11.10.4 <u>Loop Control Processing</u>. Loop control processing determines if further execution of the range of the DO-loop is required. The iteration count is tested. If it is not zero, execution of the first statement in the range of the DO-loop begins. If the iteration count is zero, the DO-loop becomes inactive. If, as a result, all of the DO-loops sharing the terminal statement of this DO-loop are inactive, normal execution continues with execution of the next executable statement following the terminal statement. However, if some of the DO-loops sharing the terminal statement are active, execution continues with incrementation processing, as described in 11.10.7.

11.10.5 <u>Execution of the Range</u>. Statements in the range of a DO-loop are executed until the terminal statement is reached. Except by the incrementation described in 11.10.7, the DO-variable of the DO-loop may neither be redefined nor become undefined during execution of the range of the DO-loop.

11.10.6 <u>Terminal Statement Execution</u>. Execution of the terminal statement occurs as a result of the normal execution sequence or as a result of transfer of control, subject to the restrictions in 11.10.8. Unless execution of the terminal statement results in a transfer of control, execution then continues with incrementation processing, as described in 11.10.7.

11.10.7 <u>Incrementation Processing</u>. Incrementation processing has the effect of the following steps performed in sequence:

(1) The DO-variable, the iteration count, and the incrementation parameter of the active DO-loop whose DO statement was most recently executed, are selected for processing.

(2) The value of the DO-variable is incremented by the value of the incrementation parameter $\underline{m}_3$.

(3) The iteration count is decremented by one.

(4) Execution continues with loop control processing (11.10.4) of the same DO-loop whose iteration count was decremented.

For example:

```
      N=0
      DO 100 I=1,10
      J=I
      DO 100 K=1,5
      L=K
  100 N=N+1
  101 CONTINUE
```

After execution of these statements and at the execution of the CONTINUE statement, I=11, J=10, K=6, L=5, and N=50.

statement are active, execution continues with incrementation processing, as described in 11.10.7.

11.10.5 Execution of the Range. Statements in the range of a DO-loop are executed until the terminal statement is reached. Except by the incrementation described in 11.10.7, the DO-variable of the DO-loop may neither be redefined nor become undefined during execution of the range of the DO-loop.

11.10.6 Terminal Statement Execution. Execution of the terminal statement occurs as a result of the normal execution sequence or as a result of transfer of control, subject to the restrictions in 11.10.8. Unless execution of the terminal statement results in a transfer of control, execution then continues with incrementation processing, as described in 11.10.7.

11.10.7 Incrementation Processing. Incrementation processing has the effect of the following steps performed in sequence:

(1) The DO-variable, the iteration count, and the incrementation parameter of the active DO-loop whose DO statement was most recently executed, are selected for processing.

(2) The value of the DO-variable is incremented by the value of the incrementation parameter $\underline{m}_3$.

(3) The iteration count is decremented by one.

(4) Execution continues with loop control processing (11.10.4) of the same DO-loop whose iteration count was decremented.

For example:

```
      N=0
      DO 100 I=1,10
      J=I
      DO 100 K=1,5
      L=K
  100 N=N+1
  101 CONTINUE
```

After execution of these statements and at the execution of the CONTINUE statement, I=11, J=10, K=6, L=5, and N=50.

Also consider the following example:

```
          N=0
          DO 200 I=1,10
          J=I
          DO 200 K=5,1
          L=K
      200 N=N+1
      201 CONTINUE
```

After execution of these statements and at the execution of the CONTINUE statement, I=11, J=10, K=5, and N=0. L is not defined by these statements.

11.10.8 Transfer into the Range of a DO-Loop. Transfer of control into the range of a DO-loop from outside the range is not permitted.

11.11 CONTINUE Statement

The form of a CONTINUE statement is:

CONTINUE

Execution of a CONTINUE statement has no effect.

If the CONTINUE statement is the terminal statement of a DO-loop, the next statement executed depends on the result of the DO-loop incrementation processing (11.10.7).

11.12 STOP Statement

The form of a STOP statement is:

STOP [*n*]

where *n* is a string of not more than five digits, or is a character constant.

Execution of a STOP statement causes termination of execution of the executable program. At the time of termination, the digit string or character constant is accessible.

11.13 PAUSE Statement

The form of a PAUSE statement is:

PAUSE [*n*]

where *n* is a string of not more than five digits, or is a character constant.

Execution of a PAUSE statement causes a cessation of execution of the executable program. Execution must be resumable. At the time of cessation of execution, the digit

Also consider the following example:

```
          N=0
          DO 200 I=1,10
          J=I
          DO 200 K=5,1
          L=K
      200 N=N+1
      201 CONTINUE
```

After execution of these statements and at the execution of the CONTINUE statement, I=11, J=10, K=5, and N=0. L is not defined by these statements.

11.10.8 Transfer into the Range of a DO-Loop. Transfer of control into the range of a DO-loop from outside the range is not permitted.

11.11 CONTINUE Statement

The form of a CONTINUE statement is:

CONTINUE

Execution of a CONTINUE statement has no effect.

If the CONTINUE statement is the terminal statement of a DO-loop, the next statement executed depends on the result of the DO-loop incrementation processing (11.10.7).

11.12 STOP Statement

The form of a STOP statement is:

STOP [n]

where n is a string of not more than five digits, or is a character constant.

Execution of a STOP statement causes termination of execution of the executable program. At the time of termination, the digit string or character constant is accessible.

11.13 PAUSE Statement

The form of a PAUSE statement is:

PAUSE [n]

where n is a string of not more than five digits, or is a character constant.

Execution of a PAUSE statement causes a cessation of execution of the executable program. Execution must be resumable. At the time of cessation of execution, the digit string or character constant is accessible. Resumption of execution is not under control of the program. If execution is resumed, the execution sequence continues as though a CONTINUE statement were executed.

11.14 END Statement

The END statement indicates the end of the sequence of statements and comment lines of a program unit (3.5). If executed in a function or subroutine subprogram, it has the effect of a RETURN statement (15.8). If executed in a main program, it terminates the execution of the executable program.

The form of an END statement is:

```
END
```

An END statement is written only in columns 7 through 72 of an initial line. An END statement must not be continued. No other statement in a program unit may have an initial line that appears to be an END statement.

The last line of every program unit must be an END statement.

string or character constant is accessible. Resumption of execution is not under control of the program. If execution is resumed, the execution sequence continues as though a CONTINUE statement were executed.

11.14 END Statement

The END statement indicates the end of the sequence of statements and comment lines of a program unit (3.5). If executed in a function or subroutine subprogram, it has the effect of a RETURN statement (15.8). If executed in a main program, it terminates the execution of the executable program.

The form of an END statement is:

```
END
```

An END statement is written only in columns 7 through 72 of an initial line. An END statement must not be continued. No other statement in a program unit may have an initial line that appears to be an END statement.

The last line of every program unit must be an END statement.

12. INPUT/OUTPUT STATEMENTS

Input statements provide the means of transferring data from external media to internal storage or from an internal file to internal storage. This process is called reading. Output statements provide the means of transferring data from internal storage to external media or from internal storage to an internal file. This process is called writing. Some input/output statements specify that editing of the data is to be performed.

In addition to the statements that transfer data, there are auxiliary input/output statements to manipulate the external medium, or to describe the properties of the connection to the external medium.

There are six input/output statements:

(1) READ

(2) WRITE

(3) OPEN

(4) BACKSPACE

(5) ENDFILE

(6) REWIND

The READ and WRITE statements are data transfer input/output statements (12.8). The OPEN, BACKSPACE, ENDFILE, and REWIND statements are auxiliary input/output statements (12.10). The BACKSPACE, ENDFILE, and REWIND statements are file positioning input/output statements (12.10.4).

12.1 Records

A record is a sequence (2.1) of values or a sequence of characters. For example, a punched card is usually considered to be a record. However, a record does not necessarily correspond to a physical entity. There are three kinds of records:

(1) Formatted

(2) Unformatted

(3) Endfile

12. INPUT/OUTPUT STATEMENTS

Input statements provide the means of transferring data from external media to internal storage or from an internal file to internal storage. This process is called <u>reading</u>. Output statements provide the means of transferring data from internal storage to external media or from internal storage to an internal file. This process is called <u>writing</u>. Some input/output statements specify that editing of the data is to be performed.

In addition to the statements that transfer data, there are auxiliary input/output statements to manipulate the external medium, or to inquire about or describe the properties of the connection to the external medium.

There are nine input/output statements:

(1) READ

(2) WRITE

(3) PRINT

(4) OPEN

(5) CLOSE

(6) INQUIRE

(7) BACKSPACE

(8) ENDFILE

(9) REWIND

The READ, WRITE, and PRINT statements are <u>data transfer input/output statements</u> (12.8). The OPEN, CLOSE, INQUIRE, BACKSPACE, ENDFILE, and REWIND statements are <u>auxiliary input/output statements</u> (12.10). The BACKSPACE, ENDFILE, and REWIND statements are <u>file positioning input/output statements</u> (12.10.4).

12.1 <u>Records</u>

A <u>record</u> is a sequence (2.1) of values or a sequence of characters. For example, a punched card is usually considered to be a record. However, a record does not necessarily correspond to a physical entity. There are three kinds of records:

(1) Formatted

(2) Unformatted

(3) Endfile

12.1.1 Formatted Record. A formatted record consists of a sequence of characters that are capable of representation in the processor. The length of a formatted record is measured in characters and depends primarily on the number of characters put into the record when it is written. However, it may depend on the processor and the external medium. The length may be zero. Formatted records may be read or written only by formatted input/output statements (12.8.1).

Formatted records may be prepared by some means other than FORTRAN; for example, by some manual input device.

12.1.2 Unformatted Record. An unformatted record consists of a sequence of values in a processor-dependent form and may contain both character and noncharacter data or may contain no data. The length of an unformatted record is measured in processor-dependent units and depends on the output list (12.8.2) used when it is written, as well as on the processor and the external medium. The length may be zero.

Unformatted records may be read or written only by unformatted input/output statements (12.8.1).

12.1.3 Endfile Record. An endfile record is written by an ENDFILE statement. An endfile record may occur only as the last record of a file. An endfile record does not have a length property.

12.2 Files

A file is a sequence (2.1) of records.

There are two kinds of files:

(1) External

(2) Internal

12.2.1 File Existence. At any given time, there is a processor-determined set of files that are said to exist for an executable program. A file may be known to the processor, yet not exist for an executable program at a particular time. For example, security reasons may prevent a file from existing for an executable program. A file may exist and contain no records; an example is a newly created file not yet written.

To create a file means to cause a file to exist that did not previously exist. To delete a file means to terminate the existence of the file.

All input/output statements may refer to files that exist. The OPEN, WRITE, and ENDFILE statements may also refer to files that do not exist.

12.1.1 Formatted Record. A formatted record consists of a sequence of characters that are capable of representation in the processor. The length of a formatted record is measured in characters and depends primarily on the number of characters put into the record when it is written. However, it may depend on the processor and the external medium. The length may be zero. Formatted records may be read or written only by formatted input/output statements (12.8.1).

Formatted records may be prepared by some means other than FORTRAN; for example, by some manual input device.

12.1.2 Unformatted Record. An unformatted record consists of a sequence of values in a processor-dependent form and may contain both character and noncharacter data or may contain no data. The length of an unformatted record is measured in processor-dependent units and depends on the output list (12.8.2) used when it is written, as well as on the processor and the external medium. The length may be zero.

Unformatted records may be read or written only by unformatted input/output statements (12.8.1).

12.1.3 Endfile Record. An endfile record is written by an ENDFILE statement. An endfile record may occur only as the last record of a file. An endfile record does not have a length property.

12.2 Files

A file is a sequence (2.1) of records.

There are two kinds of files:

(1) External

(2) Internal

12.2.1 File Existence. At any given time, there is a processor-determined set of files that are said to exist for an executable program. A file may be known to the processor, yet not exist for an executable program at a particular time. For example, security reasons may prevent a file from existing for an executable program. A file may exist and contain no records; an example is a newly created file not yet written.

To create a file means to cause a file to exist that did not previously exist. To delete a file means to terminate the existence of the file.

All input/output statements may refer to files that exist. The INQUIRE, OPEN, CLOSE, WRITE, PRINT, and ENDFILE statements may also refer to files that do not exist.

12.2.2 *File Properties*. At any given time, there is a processor-determined *set of allowed access methods*, a processor-determined *set of allowed forms*, and a processor-determined *set of allowed record lengths* for a file.

File names are not included in the subset.

12.2.3 *File Position*. A file that is connected to a unit (12.3) has a position property. Execution of certain input/output statements affects the position of a file. Certain circumstances can cause the position of a file to become indeterminate.

The *initial point* of a file is the position just before the first record. The *terminal point* is the position just after the last record.

If a file is positioned within a record, that record is the *current record*; otherwise, there is no current record.

Let n be the number of records in the file. If $1 < i \le n$ and a file is positioned within the ith record or between the (i-1)th record and the ith record, the (i-1)th record is the *preceding record*. If $n \ge 1$ and a file is positioned at its terminal point, the preceding record is the nth and last record. If n=0 or if a file is positioned at its initial point or within the first record, there is no preceding record.

If $1 \le i < n$ and a file is positioned within the ith record or between the ith and (i+1)th record, the (i+1)th record is the *next record*. If $n \ge 1$ and the file is positioned at its initial point, the first record is the next record. If n=0 or if a file is positioned at its terminal point or within the nth and last record, there is no next record.

12.2.4 *File Access*. There are two methods of accessing the records of an external file: sequential and direct. Some files may have more than one allowed access method; other files may be restricted to one access method. For example, a processor may allow only sequential access to a file on magnetic tape. Thus, the set of allowed access methods depends on the file and the processor.

The method of accessing the file is determined when the file is connected to a unit (12.3.2).

An internal file must be accessed sequentially.

12.2.4.1 *Sequential Access*. When connected for sequential access, a file has the following properties:

12.2.2 File Properties. At any given time, there is a processor-determined set of allowed access methods, a processor-determined set of allowed forms, and a processor-determined set of allowed record lengths for a file.

A file may have a name; a file that has a name is called a named file. The name of a named file is a character string. The set of allowable names is processor dependent and may be empty.

12.2.3 File Position. A file that is connected to a unit (12.3) has a position property. Execution of certain input/output statements affects the position of a file. Certain circumstances can cause the position of a file to become indeterminate.

The initial point of a file is the position just before the first record. The terminal point is the position just after the last record.

If a file is positioned within a record, that record is the current record; otherwise, there is no current record.

Let n be the number of records in the file. If $1 < i \le n$ and a file is positioned within the ith record or between the (i-1)th record and the ith record, the (i-1)th record is the preceding record. If $n \ge 1$ and a file is positioned at its terminal point, the preceding record is the nth and last record. If n=0 or if a file is positioned at its initial point or within the first record, there is no preceding record.

If $1 \le i < n$ and a file is positioned within the ith record or between the ith and (i+1)th record, the (i+1)th record is the next record. If $n \ge 1$ and the file is positioned at its initial point, the first record is the next record. If n=0 or if a file is positioned at its terminal point or within the nth and last record, there is no next record.

12.2.4 File Access. There are two methods of accessing the records of an external file: sequential and direct. Some files may have more than one allowed access method; other files may be restricted to one access method. For example, a processor may allow only sequential access to a file on magnetic tape. Thus, the set of allowed access methods depends on the file and the processor.

The method of accessing the file is determined when the file is connected to a unit (12.3.2).

An internal file must be accessed sequentially.

12.2.4.1 Sequential Access. When connected for sequential access, a file has the following properties:

(1) The order of the records is the order in which they were written if the direct access method is not a member of the set of allowed access methods for the file. If the direct access method is also a member of the set of allowed access methods for the file, the order of the records is the same as that specified for direct access (12.2.4.2). The first record accessed by sequential access is the record whose record number is 1 for direct access. The second record accessed by sequential access is the record whose record number is 2 for direct access, etc. A record that has not been written since the file was created must not be read.

(2) The records of the file are either all formatted or all unformatted, except that the last record of the file may be an endfile record.

(3) The records of the file must not be read or written by direct access input/output statements (12.8.1).

12.2.4.2 <u>Direct Access</u>. When connected for direct access, a file has the following properties:

(1) The order of the records is the order of their record numbers. The records may be read or written in any order.

(2) The records of the file are all unformatted. If the sequential access method is also a member of the set of allowed access methods for the file, its endfile record, if any, is not considered to be part of the file while it is connected for direct access. If the sequential access method is not a member of the set of allowed access methods for the file, the file must not contain an endfile record.

(3) Reading and writing records is accomplished only by direct access input/output statements (12.8.1).

(4) All records of the file have the same length.

(5) Each record of the file is uniquely identified by a positive integer called the <u>record number</u>. The record number of a record is specified when the record is written. Once established, the record number of a record can never be changed. Note that a record may not be deleted; however, a record may be rewritten.

(6) Records need not be read or written in the order of their record numbers. Any record may be written into the file while it is connected (12.3.2) to a unit. For example, it is permissible to write record 3, even though records 1 and 2 have not been written.

(1) The order of the records is the order in which they were written if the direct access method is not a member of the set of allowed access methods for the file. If the direct access method is also a member of the set of allowed access methods for the file, the order of the records is the same as that specified for direct access (12.2.4.2). The first record accessed by sequential access is the record whose record number is 1 for direct access. The second record accessed by sequential access is the record whose record number is 2 for direct access, etc. A record that has not been written since the file was created must not be read.

(2) The records of the file are either all formatted or all unformatted, except that the last record of the file may be an endfile record.

(3) The records of the file must not be read or written by direct access input/output statements (12.8.1).

12.2.4.2 <u>Direct Access</u>. When connected for direct access, a file has the following properties:

(1) The order of the records is the order of their record numbers. The records may be read or written in any order.

(2) The records of the file are either all formatted or all unformatted. If the sequential access method is also a member of the set of allowed access methods for the file, its endfile record, if any, is not considered to be part of the file while it is connected for direct access. If the sequential access method is not a member of the set of allowed access methods for the file, the file must not contain an endfile record.

(3) Reading and writing records is accomplished only by direct access input/output statements (12.8.1).

(4) All records of the file have the same length.

(5) Each record of the file is uniquely identified by a positive integer called the <u>record number</u>. The record number of a record is specified when the record is written. Once established, the record number of a record can never be changed. Note that a record may not be deleted; however, a record may be rewritten.

(6) Records need not be read or written in the order of their record numbers. Any record may be written into the file while it is connected (12.3.2) to a unit. For example, it is permissible to write record 3, even though records 1 and 2 have not been written.

Any record may be read from the file while it is connected to a unit, provided that the record was written since the file was created.

12.2.5 Internal Files. Internal files provide a means of transferring and converting data from internal storage to internal storage.

12.2.5.1 Internal File Properties. An internal file has the following properties:

(1) The file is a character variable or character array element.

(2) A record of an internal file is a character variable or character array element.

(3) The file consists of a single record whose length is the same as the length of the variable or array element.

(4) The variable or array element that is the record of the internal file becomes defined by writing the record. If the number of characters written in a record is less than the length of the record, the remaining portion of the record is filled with blanks.

(5) A record may be read only if the variable or array element that is the record is defined.

(6) A variable or array element that is a record of an internal file may become defined (or undefined) by means other than an output statement. For example, the variable or array element may become defined by a character assignment statement.

(7) An internal file is always positioned at the beginning of the record prior to data transfer.

Any record may be read from the file while it is connected to a unit, provided that the record was written since the file was created.

(7) The records of the file must not be read or written using list-directed formatting.

12.2.5 <u>Internal Files</u>. Internal files provide a means of transferring and converting data from internal storage to internal storage.

12.2.5.1 <u>Internal File Properties</u>. An internal file has the following properties:

(1) The file is a character variable, character array element, character array, or character substring.

(2) A record of an internal file is a character variable, character array element, or character substring.

(3) If the file is a character variable, character array element, or character substring, it consists of a single record whose length is the same as the length of the variable, array element, or substring, respectively. If the file is a character array, it is treated as a sequence of character array elements. Each array element is a record of the file. The ordering of the records of the file is the same as the ordering of the array elements in the array (5.2.4). Every record of the file has the same length, which is the length of an array element in the array.

(4) The variable, array element, or substring that is the record of the internal file becomes defined by writing the record. If the number of characters written in a record is less than the length of the record, the remaining portion of the record is filled with blanks.

(5) A record may be read only if the variable, array element, or substring that is the record is defined.

(6) A variable, array element, or substring that is a record of an internal file may become defined (or undefined) by means other than an output statement. For example, the variable, array element, or substring may become defined by a character assignment statement.

(7) An internal file is always positioned at the beginning of the first record prior to data transfer.

12.2.5.2 Internal File Restrictions. An internal file has the following restrictions:

(1) Reading and writing records is accomplished only by sequential access formatted input/output statements (12.8.1).

(2) An auxiliary input/output statement must not specify an internal file.

12.3 Units

A unit is a means of referring to a file.

12.3.1 Unit Existence. At any given time, there is a processor-determined set of units that are said to exist for an executable program.

All input/output statements may refer to units that exist.

12.3.2 Connection of a Unit. A unit has a property of being connected or not connected. If connected, it refers to a file. A unit may become connected by preconnection or by the execution of an OPEN statement. The property of connection is symmetric: if a unit is connected to a file, the file is connected to the unit.

Preconnection means that the unit is connected to a file at the beginning of execution of the executable program and therefore may be referenced by input/output statements without the prior execution of an OPEN statement.

All input/output statements except OPEN must reference a unit that is connected to a file and thereby make use of or affect that file.

A file may be connected and not exist. An example is a preconnected new file.

A unit must not be connected to more than one file at the same time, and a file must not be connected to more than one unit at the same time.

12.2.5.2 Internal File Restrictions. An internal file has the following restrictions:

(1) Reading and writing records is accomplished only by sequential access formatted input/output statements (12.8.1) that do not specify list-directed formatting.

(2) An auxiliary input/output statement must not specify an internal file.

12.3 Units

A *unit* is a means of referring to a file.

12.3.1 Unit Existence. At any given time, there is a processor-determined set of units that are said to *exist* for an executable program.

All input/output statements may refer to units that exist. The INQUIRE and CLOSE statements may also refer to units that do not exist.

12.3.2 Connection of a Unit. A unit has a property of being connected or not connected. If connected, it refers to a file. A unit may become connected by preconnection or by the execution of an OPEN statement. The property of connection is symmetric: if a unit is connected to a file, the file is connected to the unit.

Preconnection means that the unit is connected to a file at the beginning of execution of the executable program and therefore may be referenced by input/output statements without the prior execution of an OPEN statement.

All input/output statements except OPEN, CLOSE, and INQUIRE must reference a unit that is connected to a file and thereby make use of or affect that file.

A file may be connected and not exist. An example is a preconnected new file.

A unit must not be connected to more than one file at the same time, and a file must not be connected to more than one unit at the same time. However, means are provided to change the status of a unit and to connect a unit to a different file.

After a unit has been disconnected by the execution of a CLOSE statement, it may be connected again within the same executable program to the same file or a different file. After a file has been disconnected by the execution of a CLOSE statement, it may be connected again within the same executable program to the same unit or a different unit. Note, however, that the only means to refer to a file that has been disconnected is by its name in an OPEN or INQUIRE

12.3.3 Unit Specifier and Identifier. The form of a unit specifier is:

u

where u is an external unit identifier or an internal file identifier.

An external unit identifier is used to refer to an external file. An internal file identifier is used to refer to an internal file.

An external unit identifier is one of the following:

(1) An integer constant i or integer variable i whose value must be zero or positive

(2) An asterisk, identifying a particular processor-determined external unit that is preconnected for formatted sequential access (12.9.2)

The external unit identified by the value of i is the same external unit in all program units of the executable program. In the example:

```
SUBROUTINE A
READ (6) X


SUBROUTINE B
N=6
REWIND N
```

the value 6 used in both program units identifies the same external unit.

An external unit identifier in an auxiliary input/output statement (12.10) must not be an asterisk.

An internal file identifier is the name of a character variable or character array element.

The unit specifier must be the first item in a list of specifiers.

12.4 Format Specifier and Identifier

The form of a format specifier is:

f

statement. Therefore, there may be no means of reconnecting an unnamed file once it is disconnected.

12.3.3 Unit Specifier and Identifier. The form of a *unit specifier* is:

```
[UNIT =] u
```

where *u* is an external unit identifier or an internal file identifier.

An external unit identifier is used to refer to an external file. An internal file identifier is used to refer to an internal file.

An *external unit identifier* is one of the following:

(1) An integer expression *i* whose value must be zero or positive

(2) An asterisk, identifying a particular processor-determined external unit that is preconnected for formatted sequential access (12.9.2)

The external unit identified by the value of *i* is the same external unit in all program units of the executable program. In the example:

```
SUBROUTINE A
READ (6) X


SUBROUTINE B
N=6
REWIND N
```

the value 6 used in both program units identifies the same external unit.

An external unit identifier in an auxiliary input/output statement (12.10) must not be an asterisk.

An *internal file identifier* is the name of a character variable, character array, character array element, or character substring.

If the optional characters UNIT= are omitted from the unit specifier, the unit specifier must be the first item in a list of specifiers.

12.4 Format Specifier and Identifier

The form of a *format specifier* is:

```
[FMT =] f
```

where _f_ is a format identifier.

A _format identifier_ identifies a format. A format identifier must be one of the following:

(1) The statement label of a FORMAT statement that appears in the same program unit as the format identifier.

(2) An integer variable name that has been assigned the statement label of a FORMAT statement that appears in the same program unit as the format identifier (10.3).

(3) A character constant (13.1.2).

If present, the format specifier must be the second item in the control information list and the first item must be the unit specifier.

12.5 Record Specifier

The form of a _record specifier_ is:

REC = _rn_

where _rn_ is an integer constant or integer variable whose value is positive. It specifies the number of the record that is to be read or written in a file connected for direct access.

12.6 Error and End-of-File Conditions

The set of input/output error conditions is processor dependent.

An end-of-file condition exists if either of the following events occurs:

(1) An endfile record is encountered during the reading of a file connected for sequential access. In this case, the file is positioned after the endfile record.

(2) An attempt is made to read a record beyond the end of an internal file.

where _f_ is a format identifier.

A _format identifier_ identifies a format. A format identifier must be one of the following:

(1) The statement label of a FORMAT statement that appears in the same program unit as the format identifier.

(2) An integer variable name that has been assigned the statement label of a FORMAT statement that appears in the same program unit as the format identifier (10.3).

(3) A character array name (13.1.2).

(4) Any character expression except a character expression involving concatenation of an operand whose length specification is an asterisk in parentheses unless the operand is the symbolic name of a constant. Note that a character constant is permitted.

(5) An asterisk, specifying list-directed formatting.

If the optional characters FMT= are omitted from the format specifier, the format specifier must be the second item in the control information list and the first item must be the unit specifier without the optional characters UNIT=.

12.5 Record Specifier

The form of a _record specifier_ is:

REC = _rn_

where _rn_ is an integer expression whose value is positive. It specifies the number of the record that is to be read or written in a file connected for direct access.

12.6 Error and End-of-File Conditions

The set of input/output error conditions is processor dependent.

An end-of-file condition exists if either of the following events occurs:

(1) An endfile record is encountered during the reading of a file connected for sequential access. In this case, the file is positioned after the endfile record.

(2) An attempt is made to read a record beyond the end of an internal file.

If an end-of-file condition occurs during execution of a READ statement, execution of the READ statement terminates and the entities specified by the input list and implied-DO-variables in the input list become undefined. Note that variables appearing only in subscripts and implied-DO parameters in an input list do not become undefined when the entities specified by the list become undefined.

If an error condition occurs during execution of an output statement, execution of the output statement terminates and implied-DO-variables in the output list become undefined.

If an error condition occurs during execution of an input/output statement, or if an end-of-file condition occurs during execution of a READ statement that does not contain an end-of-file specifier (12.7.2), execution of the executable program is terminated.

12.7 Input/Output Status, Error, and End-of-File Specifiers

The input/output status specifier is not included in the subset.

12.7.1 Error Specifier. The error specifier is not included in the subset.

If an error condition occurs during execution of an input/output statement, execution of the input/output statement terminates and the position of the file becomes indeterminate.

If an error condition or an end-of-file condition occurs during execution of a READ statement, execution of the READ statement terminates and the entities specified by the input list and implied-DO-variables in the input list become undefined. Note that variables and array elements appearing only in subscripts, substring expressions, and implied-DO parameters in an input list do not become undefined when the entities specified by the list become undefined.

If an error condition occurs during execution of an output statement, execution of the output statement terminates and implied-DO-variables in the output list become undefined.

If an error condition occurs during execution of an input/output statement that contains neither an input/output status specifier (12.7) nor an error specifier (12.7.1), or if an end-of-file condition occurs during execution of a READ statement that contains neither an input/output status specifier nor an end-of-file specifier (12.7.2), execution of the executable program is terminated.

12.7 Input/Output Status, Error, and End-of-File Specifiers

The form of an input/output status specifier is:

IOSTAT = *ios*

where *ios* is an integer variable or integer array element.

Execution of an input/output statement containing this specifier causes *ios* to become defined:

(1) with a zero value if neither an error condition nor an end-of-file condition is encountered by the processor,

(2) with a processor-dependent positive integer value if an error condition is encountered, or

(3) with a processor-dependent negative integer value if an end-of-file condition is encountered and no error condition is encountered.

12.7.1 Error Specifier.
The form of an error specifier is:

ERR = *s*

where *s* is the statement label of an executable statement that appears in the same program unit as the error specifier.

12.7.2 End-of-File Specifier. The form of an end-of-file specifier is:

END = s

where s is the statement label of an executable statement that appears in the same program unit as the end-of-file specifier.

If a READ statement contains an end-of-file specifier and the processor encounters an end-of-file condition and no error condition during execution of the statement:

(1) execution of the READ statement terminates, and

(2) execution continues with the statement labeled s.

12.8 READ, WRITE, and PRINT Statements

The READ statement is the data transfer input statement. The WRITE statement is the data transfer output statement. The forms of the data transfer input/output statements are:

READ (cilist) [iolist]

WRITE (cilist) [iolist]

where: cilist is a control information list (12.8.1) that includes:

If an input/output statement contains an error specifier and the processor encounters an error condition during execution of the statement:

(1) execution of the input/output statement terminates,

(2) the position of the file specified in the input/output statement becomes indeterminate,

(3) if the input/output statement contains an input/output status specifier (12.7), the variable or array element _ios_ becomes defined with a processor-dependent positive integer value, and

(4) execution continues with the statement labeled _s_.

12.7.2 _End-of-File Specifier_. The form of an _end-of-file specifier_ is:

    END = s

where _s_ is the statement label of an executable statement that appears in the same program unit as the end-of-file specifier.

If a READ statement contains an end-of-file specifier and the processor encounters an end-of-file condition and no error condition during execution of the statement:

(1) execution of the READ statement terminates,

(2) if the READ statement contains an input/output status specifier (12.7), the variable or array element _ios_ becomes defined with a processor-dependent negative integer value, and

(3) execution continues with the statement labeled _s_.

12.8 READ, WRITE, and PRINT Statements

The READ statement is the data transfer input statement. The WRITE and PRINT statements are the data transfer output statements. The forms of the data transfer input/output statements are:

    READ (cilist) [iolist]

    READ f [,iolist]

    WRITE (cilist) [iolist]

    PRINT f [,iolist]

where: _cilist_ is a control information list (12.8.1) that includes:

(1) A reference to the source or destination of the data to be transferred

(2) Optional specification of editing processes

(3) Optional specifiers that determine the execution sequence on the occurrence of certain events

(4) Optional specification to identify a record

<u>iolist</u> is an input/output list (12.8.2) specifying the data to be transferred

The PRINT statement and READ statement without a <u>cilist</u> are not included in the subset.

12.8.1 <u>Control Information List</u>. A <u>control information list</u>, <u>cilist</u>, is a list (2.10) whose list items may be any of the following:

```
u
f
REC = rn
END = s
```

A control information list must contain exactly one unit specifier (12.3.3), at most one format specifier (12.4), at most one record specifier (12.5), and at most one end-of-file specifier (12.7.2).

If the control information list contains a format specifier, the statement is a <u>formatted input/output statement</u>; otherwise, it is an <u>unformatted input/output statement</u>.

If the control information list contains a record specifier, the statement is a <u>direct access input/output statement</u>; otherwise, it is a <u>sequential access input/output statement</u>.

The unit specifier must be the first item in the control information list.

If present, the format specifier must be the second item in the control information list and the first item must be the unit specifier.

(1) A reference to the source or destination of the data to be transferred

(2) Optional specification of editing processes

(3) Optional specifiers that determine the execution sequence on the occurrence of certain events

(4) Optional specification to identify a record

(5) Optional specification to provide the return of the input/output status

f is a format identifier (12.4)

iolist is an input/output list (12.8.2) specifying the data to be transferred

12.8.1 Control Information List. A control information list, cilist, is a list (2.10) whose list items may be any of the following:

```
[UNIT =] u
[FMT =] f
REC = rn
IOSTAT = ios
ERR = s
END = s
```

A control information list must contain exactly one unit specifier (12.3.3), at most one format specifier (12.4), at most one record specifier (12.5), at most one input/output status specifier (12.7), at most one error specifier (12.7.1), and at most one end-of-file specifier (12.7.2).

If the control information list contains a format specifier, the statement is a formatted input/output statement; otherwise, it is an unformatted input/output statement.

If the control information list contains a record specifier, the statement is a direct access input/output statement; otherwise, it is a sequential access input/output statement.

If the optional characters UNIT= are omitted from the unit specifier, the unit specifier must be the first item in the control information list.

If the optional characters FMT= are omitted from the format specifier, the format specifier must be the second item in

A control information list must not contain both a record specifier and an end-of-file specifier, or both a format specifier and a record specifier.

In a WRITE statement, the control information list must not contain an end-of-file specifier.

If the unit specifier specifies an internal file, the control information list must contain a format identifier and must not contain a record specifier.

12.8.2 Input/Output List. An input/output list, iolist, specifies the entities whose values are transferred by a data transfer input/output statement.

An input/output list is a list (2.10) of input/output list items and implied-DO lists (12.8.2.3). An input/output list item is either an input list item or an output list item.

If an array name appears as an input/output list item, it is treated as if all of the elements of the array were specified in the order given by array element ordering (5.2.4). The name of an assumed-size dummy array must not appear as an input/output list item.

12.8.2.1 Input List Items. An input list item must be one of the following:

(1) A variable name

(2) An array element name

(3) An array name

Only input list items may appear as input/output list items in an input statement.

12.8.2.2 Output List Items. An output list item must be one of the following:

(1) A variable name

(2) An array element name

(3) An array name

the control information list and the first item must be the unit specifier without the optional characters UNIT=.

A control information list must not contain both a record specifier and an end-of-file specifier.

If the format identifier is an asterisk, the statement is a list-directed input/output statement and a record specifier must not be present.

In a WRITE statement, the control information list must not contain an end-of-file specifier.

If the unit specifier specifies an internal file, the control information list must contain a format identifier other than an asterisk and must not contain a record specifier.

12.8.2 Input/Output List. An input/output list, iolist, specifies the entities whose values are transferred by a data transfer input/output statement.

An input/output list is a list (2.10) of input/output list items and implied-DO lists (12.8.2.3). An input/output list item is either an input list item or an output list item.

If an array name appears as an input/output list item, it is treated as if all of the elements of the array were specified in the order given by array element ordering (5.2.4). The name of an assumed-size dummy array must not appear as an input/output list item.

12.8.2.1 Input List Items. An input list item must be one of the following:

(1) A variable name

(2) An array element name

(3) A character substring name

(4) An array name

Only input list items may appear as input/output list items in an input statement.

12.8.2.2 Output List Items. An output list item must be one of the following:

(1) A variable name

(2) An array element name

(3) A character substring name

(4) An array name

12.8.2.3 Implied-DO List. An implied-DO list is of the form:

$$(\underline{dlist}, \underline{i} = \underline{e}_1, \underline{e}_2 \, [, \underline{e}_3 \,] \,)$$

where: $\underline{i}$, $\underline{e}_1$, $\underline{e}_2$, and $\underline{e}_3$ are as specified for the DO statement (11.10)

dlist is an input/output list

The range of an implied-DO list is the list dlist. Note that dlist may contain implied-DO lists. The iteration count and the values of the DO-variable $\underline{i}$ are established from $\underline{e}_1$, $\underline{e}_2$, and $\underline{e}_3$ exactly as for a DO-loop. In an input statement, the DO-variable $\underline{i}$, or an associated entity, must not appear as an input list item in dlist. When an implied-DO list appears in an input/output list, the list items in dlist are specified once for each iteration of the implied-DO list with appropriate substitution of values for any occurrence of the DO-variable $\underline{i}$.

12.9 Execution of a Data Transfer Input/Output Statement

The effect of executing a data transfer input/output statement must be as if the following operations were performed in the order specified:

(1) Determine the direction of data transfer

(2) Identify the unit

(3) Establish the format if any is specified

(4) Position the file prior to data transfer

(5) Transfer data between the file and the entities specified by the input/output list (if any)

(6) Position the file after data transfer

(5) Any other expression except a character expression involving concatenation of an operand whose length specification is an asterisk in parentheses unless the operand is the symbolic name of a constant

Note that a constant, an expression involving operators or function references, or an expression enclosed in parentheses may appear as an output list item but must not appear as an input list item.

12.8.2.3 Implied-DO List. An implied-DO list is of the form:

(dlist, i = e_1, e_2 [,e_3])

where: i, e_1, e_2, and e_3 are as specified for the DO statement (11.10)

dlist is an input/output list

The range of an implied-DO list is the list dlist. Note that dlist may contain implied-DO lists. The iteration count and the values of the DO-variable i are established from e_1, e_2, and e_3 exactly as for a DO-loop. In an input statement, the DO-variable i, or an associated entity, must not appear as an input list item in dlist. When an implied-DO list appears in an input/output list, the list items in dlist are specified once for each iteration of the implied-DO list with appropriate substitution of values for any occurrence of the DO-variable i.

12.9 Execution of a Data Transfer Input/Output Statement

The effect of executing a data transfer input/output statement must be as if the following operations were performed in the order specified:

(1) Determine the direction of data transfer

(2) Identify the unit

(3) Establish the format if any is specified

(4) Position the file prior to data transfer

(5) Transfer data between the file and the entities specified by the input/output list (if any)

(6) Position the file after data transfer

(7) Cause the specified integer variable or array element in the input/output status specifier (if any) to become defined

12.9.1 <u>Direction of Data Transfer</u>. Execution of a READ statement causes values to be transferred from a file to the entities specified by the input list, if one is specified.

Execution of a WRITE statement causes values to be transferred to a file from the entities specified by the output list and format specification (if any). Execution of a WRITE statement for a file that does not exist creates the file, unless an error condition occurs.

12.9.2 <u>Identifying a Unit</u>. A data transfer input/output statement includes a unit specifier that identifies an external unit or an internal file. A READ statement that contains an asterisk as the unit identifier specifies a particular processor-determined unit. A WRITE statement that contains an asterisk as the unit identifier specifies some other processor-determined unit. Thus, each data transfer input/output statement identifies an external unit or an internal file.

The unit identified by a data transfer input/output statement must be connected to a file when execution of the statement begins.

12.9.3 <u>Establishing a Format</u>. If the control information list contains a format identifier, the format specification identified by the format identifier is established.

On output, if an internal file has been specified, a format specification (13.1) that is in the file or is associated (17.1) with the file must not be specified.

12.9.4 <u>File Position Prior to Data Transfer</u>. The positioning of the file prior to data transfer depends on the method of access: sequential or direct.

If the file contains an endfile record, the file must not be positioned after the endfile record prior to data transfer.

12.9.4.1 <u>Sequential Access</u>. On input, the file is positioned at the beginning of the next record. This record becomes the current record. On output, a new record is created and becomes the last record of the file.

An internal file is always positioned at the beginning of the record of the file. This record becomes the current record.

12.9.1 Direction of Data Transfer. Execution of a READ statement causes values to be transferred from a file to the entities specified by the input list, if one is specified.

Execution of a WRITE or PRINT statement causes values to be transferred to a file from the entities specified by the output list and format specification (if any). Execution of a WRITE or PRINT statement for a file that does not exist creates the file, unless an error condition occurs.

12.9.2 Identifying a Unit. A data transfer input/output statement that contains a control information list (12.8.1) includes a unit specifier that identifies an external unit or an internal file. A READ statement that does not contain a control information list specifies a particular processor-determined unit, which is the same as the unit identified by an asterisk in a READ statement that contains a control information list. A PRINT statement specifies some other processor-determined unit, which is the same as the unit identified by an asterisk in a WRITE statement. Thus, each data transfer input/output statement identifies an external unit or an internal file.

The unit identified by a data transfer input/output statement must be connected to a file when execution of the statement begins.

12.9.3 Establishing a Format. If the control information list contains a format identifier other than an asterisk, the format specification identified by the format identifier is established. If the format identifier is an asterisk, list-directed formatting is established.

On output, if an internal file has been specified, a format specification (13.1) that is in the file or is associated (17.1) with the file must not be specified.

12.9.4 File Position Prior to Data Transfer. The positioning of the file prior to data transfer depends on the method of access: sequential or direct.

If the file contains an endfile record, the file must not be positioned after the endfile record prior to data transfer.

12.9.4.1 Sequential Access. On input, the file is positioned at the beginning of the next record. This record becomes the current record. On output, a new record is created and becomes the last record of the file.

An internal file is always positioned at the beginning of the first record of the file. This record becomes the current record.

12.9.4.2 Direct Access. For direct access, the file is positioned at the beginning of the record specified by the record specifier (12.5). This record becomes the current record.

12.9.5 Data Transfer. Data are transferred between records and entities specified by the input/output list. The list items are processed in the order of the input/output list.

All values needed to determine which entities are specified by an input/output list item are determined at the beginning of the processing of that item.

All values are transmitted to or from the entities specified by a list item prior to the processing of any succeeding list item. In the example,

```
READ (3) N, A(N)
```

two values are read; one is assigned to N, and the second is assigned to A(N) for the new value of N.

An input list item, or an entity associated with it (17.1.3), must not contain any portion of the established format specification.

If an internal file has been specified, an input/output list item must not be in the file or associated with the file.

A DO-variable becomes defined at the beginning of processing of the items that constitute the range of an implied-DO list.

On output, every entity whose value is to be transferred must be defined.

On input, an attempt to read a record of a file connected for direct access that has not previously been written causes all entities specified by the input list to become undefined.

12.9.5.1 Unformatted Data Transfer. During unformatted data transfer, data are transferred without editing between the current record and the entities specified by the input/output list. Exactly one record is read or written.

On input, the file must be positioned so that the record read is an unformatted record or an endfile record.

On input, the number of values required by the input list must be less than or equal to the number of values in the record.

On input, the type of each value in the record must agree with the type of the corresponding entity in the input list. If an entity in the input list is of type character, the

12.9.4.2 Direct Access. For direct access, the file is positioned at the beginning of the record specified by the record specifier (12.5). This record becomes the current record.

12.9.5 Data Transfer. Data are transferred between records and entities specified by the input/output list. The list items are processed in the order of the input/output list.

All values needed to determine which entities are specified by an input/output list item are determined at the beginning of the processing of that item.

All values are transmitted to or from the entities specified by a list item prior to the processing of any succeeding list item. In the example,

```
READ (3) N, A(N)
```

two values are read; one is assigned to N, and the second is assigned to A(N) for the new value of N.

An input list item, or an entity associated with it (17.1.3), must not contain any portion of the established format specification.

If an internal file has been specified, an input/output list item must not be in the file or associated with the file.

A DO-variable becomes defined at the beginning of processing of the items that constitute the range of an implied-DO list.

On output, every entity whose value is to be transferred must be defined.

On input, an attempt to read a record of a file connected for direct access that has not previously been written causes all entities specified by the input list to become undefined.

12.9.5.1 Unformatted Data Transfer. During unformatted data transfer, data are transferred without editing between the current record and the entities specified by the input/output list. Exactly one record is read or written.

On input, the file must be positioned so that the record read is an unformatted record or an endfile record.

On input, the number of values required by the input list must be less than or equal to the number of values in the record.

On input, the type of each value in the record must agree with the type of the corresponding entity in the input list, except that one complex value may correspond to two real length of the character entity must agree with the length of the character value.

On output to a file connected for direct access, the output list must not specify more values than can fit into a record.

On output, if the file is connected for direct access and the values specified by the output list do not fill the record, the remainder of the record is undefined.

If the file is connected for formatted input/output, unformatted data transfer is prohibited.

The unit specified must be an external unit.

12.9.5.2 <u>Formatted Data Transfer</u>. During formatted data transfer, data are transferred with editing between the entities specified by the input/output list and the file. The current record and possibly additional records are read or written.

On input, the file must be positioned so that the record read is a formatted record or an endfile record.

If the file is connected for unformatted input/output, formatted data transfer is prohibited.

12.9.5.2.1 <u>Using a Format Specification</u>. If a format specification has been established, format control (13.3) is initiated and editing is performed as described in 13.3 through 13.5.

On input, the input list and format specification must not require more characters from a record than the record contains.

list entities or two real values may correspond to one complex list entity. If an entity in the input list is of type character, the length of the character entity must agree with the length of the character value.

On output to a file connected for direct access, the output list must not specify more values than can fit into a record.

On output, if the file is connected for direct access and the values specified by the output list do not fill the record, the remainder of the record is undefined.

If the file is connected for formatted input/output, unformatted data transfer is prohibited.

The unit specified must be an external unit.

12.9.5.2 Formatted Data Transfer. During formatted data transfer, data are transferred with editing between the entities specified by the input/output list and the file. The current record and possibly additional records are read or written.

On input, the file must be positioned so that the record read is a formatted record or an endfile record.

If the file is connected for unformatted input/output, formatted data transfer is prohibited.

12.9.5.2.1 Using a Format Specification. If a format specification has been established, format control (13.3) is initiated and editing is performed as described in 13.3 through 13.5.

On input, the input list and format specification must not require more characters from a record than the record contains.

If the file is connected for direct access, the record number is increased by one as each succeeding record is read or written.

On output, if the file is connected for direct access or is an internal file and the characters specified by the output list and format do not fill a record, blank characters are added to fill the record.

On output, if the file is connected for direct access or is an internal file, the output list and format specification must not specify more characters for a record than can fit into the record.

12.9.5.2.2 <u>List-Directed Formatting</u>. List-directed formatting is not included in the subset.

12.9.5.2.3 <u>Printing of Formatted Records</u>. The transfer of information in a formatted record to certain devices determined by the processor is called <u>printing</u>. If a formatted record is printed, the first character of the record is not printed. The remaining characters of the record, if any, are printed in one line beginning at the left margin.

The first character of such a record determines vertical spacing as follows:

Character	Vertical Spacing Before Printing
Blank	One Line
0	Two Lines
1	To First Line of Next Page
+	No Advance

If there are no characters in the record (13.5.4), the vertical spacing is one line and no characters other than blank are printed in that line.

12.9.6 <u>File Position After Data Transfer</u>. If an end-of-file condition exists as a result of reading an endfile record, the file is positioned after the endfile record.

If no error condition or end-of-file condition exists, the file is positioned after the last record read or written and that record becomes the preceding record. A record written on a file connected for sequential access becomes the last record of the file.

If the file is positioned after the endfile record, execution of a data transfer input/output statement is prohibited. However, a BACKSPACE or REWIND statement may be used to reposition the file.

12.9.7 <u>Input/Output Status Specifier Definition</u>. The input/output status specifier is not included in the subset.

12.9.5.2.2 List-Directed Formatting. If list-directed formatting has been established, editing is performed as described in 13.6.

12.9.5.2.3 Printing of Formatted Records. The transfer of information in a formatted record to certain devices determined by the processor is called printing. If a formatted record is printed, the first character of the record is not printed. The remaining characters of the record, if any, are printed in one line beginning at the left margin.

The first character of such a record determines vertical spacing as follows:

Character	Vertical Spacing Before Printing
Blank	One Line
0	Two Lines
1	To First Line of Next Page
+	No Advance

If there are no characters in the record (13.5.4), the vertical spacing is one line and no characters other than blank are printed in that line.

A PRINT statement does not imply that printing will occur, and a WRITE statement does not imply that printing will not occur.

12.9.6 File Position After Data Transfer. If an end-of-file condition exists as a result of reading an endfile record, the file is positioned after the endfile record.

If no error condition or end-of-file condition exists, the file is positioned after the last record read or written and that record becomes the preceding record. A record written on a file connected for sequential access becomes the last record of the file.

If the file is positioned after the endfile record, execution of a data transfer input/output statement is prohibited. However, a BACKSPACE or REWIND statement may be used to reposition the file.

If an error condition exists, the position of the file is indeterminate.

12.9.7 Input/Output Status Specifier Definition. If the data transfer input/output statement contains an input/output status specifier, the integer variable or array element ios becomes defined. If no error condition or end-of-file condition exists, the value of ios is zero. If an error condition exists, the value of ios is positive. If an

12.10 Auxiliary Input/Output Statements

12.10.1 OPEN Statement. An OPEN statement may be used to connect (12.3.2) an existing file to a unit, create a file (12.2.1) that is preconnected, or create a file and connect it to a unit.

The form of an OPEN statement is:

```
OPEN (olist)
```

where *olist* is a list (2.10) of specifiers:

```
u
ACCESS = 'DIRECT'
RECL = rl
```

olist must contain exactly one external unit specifier (12.3.3) and must contain exactly one of each of the other specifiers. The specified unit is connected to a processor-determined file. (See, however, 12.10.1.1.)

The other specifiers are described as follows:

end-of-file condition exists and no error condition exists, the value of ios is negative.

12.10 Auxiliary Input/Output Statements

12.10.1 OPEN Statement. An OPEN statement may be used to connect (12.3.2) an existing file to a unit, create a file (12.2.1) that is preconnected, create a file and connect it to a unit, or change certain specifiers of a connection between a file and a unit.

The form of an OPEN statement is:

```
OPEN (olist)
```

where olist is a list (2.10) of specifiers:

```
[UNIT =] u
IOSTAT = ios
ERR = s
FILE = fin
STATUS = sta
ACCESS = acc
FORM = fm
RECL = rl
BLANK = blnk
```

olist must contain exactly one external unit specifier (12.3.3) and may contain at most one of each of the other specifiers.

The other specifiers are described as follows:

IOSTAT = ios

> is an input/output status specifier (12.7). Execution of an OPEN statement containing this specifier causes ios to become defined with a zero value if no error condition exists or with a processor-dependent positive integer value if an error condition exists.

ERR = s

> is an error specifier (12.7.1).

FILE = fin

> fin is a character expression whose value when any trailing blanks are removed is the name of the file to be connected to the specified unit. The file name must be a name that is allowed by the processor. If this specifier is omitted and the unit is not

ACCESS = 'DIRECT'

specifies the access method for the connection of the file as direct (12.2.4). For an existing file, the specified access method must be included in the set of allowed access methods for the file (12.2.4). For a new file, the processor creates the file with a set of allowed access methods that includes the specified method.

connected to a file, it becomes connected to a processor-determined file. (See also 12.10.1.1.)

STATUS = <u>sta</u>

<u>sta</u> is a character expression whose value when any trailing blanks are removed is OLD, NEW, SCRATCH, or UNKNOWN. If OLD or NEW is specified, a FILE= specifier must be given. If OLD is specified, the file must exist. If NEW is specified, the file must not exist. Successful execution of an OPEN statement with NEW specified creates the file and changes the status to OLD (12.10.1.1). If SCRATCH is specified with an unnamed file, the file is connected to the specified unit for use by the executable program but is deleted (12.2.1) at the execution of a CLOSE statement referring to the same unit or at the termination of the executable program. SCRATCH must not be specified with a named file. If UNKNOWN is specified, the status is processor dependent. If this specifier is omitted, a value of UNKNOWN is assumed.

ACCESS = <u>acc</u>

<u>acc</u> is a character expression whose value when any trailing blanks are removed is SEQUENTIAL or DIRECT. It specifies the access method for the connection of the file as being sequential or direct (12.2.4). If this specifier is omitted, the assumed value is SEQUENTIAL. For an existing file, the specified access method must be included in the set of allowed access methods for the file (12.2.4). For a new file, the processor creates the file with a set of allowed access methods that includes the specified method.

FORM = <u>fm</u>

<u>fm</u> is a character expression whose value when any trailing blanks are removed is FORMATTED or UNFORMATTED. It specifies that the file is being connected for formatted or unformatted input/output, respectively. If this specifier is omitted, a value of UNFORMATTED is assumed if the file is being connected for direct access, and a value of FORMATTED is assumed if the file is being connected for sequential access. For an existing file, the specified form must be included in the set of allowed forms for the file (12.2.2). For a new file, the processor creates the file with a set of allowed forms that includes the specified form.

RECL = rl

rl is an integer constant or integer variable whose value must be positive. It specifies the length of each record in a file being connected for direct access. The length is measured in processor-determined units. For an existing file, the value of rl must be included in the set of allowed record lengths for the file (12.2.2). For a new file, the processor creates the file with a set of allowed record lengths that includes the specified value. This specifier must be given when a file is being connected for direct access.

The unit specified must exist.

A unit may be connected by execution of an OPEN statement in any program unit of an executable program and, once connected, may be referenced in any program unit of the executable program.

12.10.1.1 Open of a Connected Unit. If a unit is connected to a file that exists, execution of an OPEN statement for that unit is not permitted.

RECL = rl

> rl is an integer expression whose value must be positive. It specifies the length of each record in a file being connected for direct access. If the file is being connected for formatted input/output, the length is the number of characters. If the file is being connected for unformatted input/output, the length is measured in processor-dependent units. For an existing file, the value of rl must be included in the set of allowed record lengths for the file (12.2.2). For a new file, the processor creates the file with a set of allowed record lengths that includes the specified value. This specifier must be given when a file is being connected for direct access; otherwise, it must be omitted.

BLANK = blnk

> blnk is a character expression whose value when any trailing blanks are removed is NULL or ZERO. If NULL is specified, all blank characters in numeric formatted input fields on the specified unit are ignored, except that a field of all blanks has a value of zero. If ZERO is specified, all blanks other than leading blanks are treated as zeros. If this specifier is omitted, a value of NULL is assumed. This specifier is permitted only for a file being connected for formatted input/output.

The unit specifier is required to appear; all other specifiers are optional, except that the record length rl must be specified if a file is being connected for direct access. Note that some of the specifications have an assumed value if they are omitted.

The unit specified must exist.

A unit may be connected by execution of an OPEN statement in any program unit of an executable program and, once connected, may be referenced in any program unit of the executable program.

12.10.1.1 Open of a Connected Unit. If a unit is connected to a file that exists, execution of an OPEN statement for that unit is permitted. If the FILE= specifier is not included in the OPEN statement, the file to be connected to the unit is the same as the file to which the unit is connected.

If the file to be connected to the unit does not exist, but is the same as the file to which the unit is preconnected, the properties specified by the OPEN statement become a part of the connection.

12.10.2 <u>CLOSE Statement</u>. The CLOSE statement is not included in the subset.

If the file to be connected to the unit is not the same as the file to which the unit is connected, the effect is as if a CLOSE statement (12.10.2) without a STATUS= specifier had been executed for the unit immediately prior to the execution of the OPEN statement.

If the file to be connected to the unit is the same as the file to which the unit is connected, only the BLANK= specifier may have a value different from the one currently in effect. Execution of the OPEN statement causes the new value of the BLANK= specifier to be in effect. The position of the file is unaffected.

If a file is connected to a unit, execution of an OPEN statement on that file and a different unit is not permitted.

12.10.2 CLOSE Statement. A CLOSE statement is used to terminate the connection of a particular file to a unit.

The form of a CLOSE statement is:

CLOSE (*cllist*)

where *cllist* is a list (2.10) of specifiers:

```
[UNIT =] u
IOSTAT = ios
ERR = s
STATUS = sta
```

cllist must contain exactly one external unit specifier (12.3.3) and may contain at most one of each of the other specifiers.

The other specifiers are described as follows:

IOSTAT = *ios*

is an input/output status specifier (12.7). Execution of a CLOSE statement containing this specifier causes *ios* to become defined with a zero value if no error condition exists or with a processor-dependent positive integer value if an error condition exists.

ERR = *s*

is an error specifier (12.7.1).

STATUS = *sta*

sta is a character expression whose value when any trailing blanks are removed is KEEP or DELETE. *sta*

12.10.2.1 Implicit Close at Termination of Execution. At termination of execution of an executable program for reasons other than an error condition, all units that are connected are closed.

12.10.3 INQUIRE Statement. The INQUIRE statement is not included in the subset.

determines the disposition of the file that is connected to the specified unit. KEEP must not be specified for a file whose status prior to execution of the CLOSE statement is SCRATCH. If KEEP is specified for a file that exists, the file continues to exist after the execution of the CLOSE statement. If KEEP is specified for a file that does not exist, the file will not exist after the execution of the CLOSE statement. If DELETE is specified, the file will not exist after execution of the CLOSE statement. If this specifier is omitted, the assumed value is KEEP, unless the file status prior to execution of the CLOSE statement is SCRATCH, in which case the assumed value is DELETE.

Execution of a CLOSE statement that refers to a unit may occur in any program unit of an executable program and need not occur in the same program unit as the execution of an OPEN statement referring to that unit.

Execution of a CLOSE statement specifying a unit that does not exist or has no file connected to it is permitted and affects no file.

After a unit has been disconnected by execution of a CLOSE statement, it may be connected again within the same executable program, either to the same file or to a different file. After a file has been disconnected by execution of a CLOSE statement, it may be connected again within the same executable program, either to the same unit or to a different unit, provided that the file still exists.

12.10.2.1 <u>Implicit Close at Termination of Execution</u>. At termination of execution of an executable program for reasons other than an error condition, all units that are connected are closed. Each unit is closed with status KEEP unless the file status prior to termination of execution was SCRATCH, in which case the unit is closed with status DELETE. Note that the effect is as though a CLOSE statement without a STATUS= specifier were executed on each connected unit.

12.10.3 <u>INQUIRE Statement</u>. An INQUIRE statement may be used to inquire about properties of a particular named file or of the connection to a particular unit. There are two forms of the INQUIRE statement: inquire by file and inquire by unit. All value assignments are done according to the rules for assignment statements.

The INQUIRE statement may be executed before, while, or after a file is connected to a unit. All values assigned by the INQUIRE statement are those that are current at the time the statement is executed.

12.10.3.1 <u>INQUIRE by File</u>. The INQUIRE statement is not included in the subset.

12.10.3.2 <u>INQUIRE by Unit</u>. The INQUIRE statement is not included in the subset.

12.10.3.3 <u>Inquiry Specifiers</u>. The INQUIRE statement is not included in the subset.

12.10.3.1 <u>INQUIRE by File</u>. The form of an INQUIRE by file statement is:

INQUIRE (<u>iflist</u>)

where <u>iflist</u> is a list (2.10) of specifiers that must contain exactly one file specifier and may contain other inquiry specifiers. The <u>iflist</u> may contain at most one of each of the inquiry specifiers described in 12.10.3.3.

The form of a file specifier is:

FILE = <u>fin</u>

where <u>fin</u> is a character expression whose value when any trailing blanks are removed specifies the name of the file being inquired about. The named file need not exist or be connected to a unit. The value of <u>fin</u> must be of a form acceptable to the processor as a file name.

12.10.3.2 <u>INQUIRE by Unit</u>. The form of an INQUIRE by unit statement is:

INQUIRE (<u>iulist</u>)

where <u>iulist</u> is a list (2.10) of specifiers that must contain exactly one external unit specifier (12.3.3) and may contain other inquiry specifiers. The <u>iulist</u> may contain at most one of each of the inquiry specifiers described in 12.10.3.3. The unit specified need not exist or be connected to a file. If it is connected to a file, the inquiry is being made about the connection and about the file connected.

12.10.3.3 <u>Inquiry Specifiers</u>. The following inquiry specifiers may be used in either form of the INQUIRE statement:

The INQUIRE statement is not included in the subset.

```
IOSTAT = ios
ERR = s
EXIST = ex
OPENED = od
NUMBER = num
NAMED = nmd
NAME = fn
ACCESS = acc
SEQUENTIAL = seq
DIRECT = dir
FORM = fm
FORMATTED = fmt
UNFORMATTED = unf
RECL = rcl
NEXTREC = nr
BLANK = blnk
```

The specifiers are described as follows:

IOSTAT = <u>ios</u>

is an input/output status specifier (12.7). Execution of an INQUIRE statement containing this specifier causes <u>ios</u> to become defined with a zero value if no error condition exists or with a processor-dependent positive integer value if an error condition exists.

ERR = <u>s</u>

is an error specifier (12.7.1).

EXIST = <u>ex</u>

<u>ex</u> is a logical variable or logical array element. Execution of an INQUIRE by file statement causes <u>ex</u> to be assigned the value true if there exists a file with the specified name; otherwise, <u>ex</u> is assigned the value false. Execution of an INQUIRE by unit statement causes <u>ex</u> to be assigned the value true if the specified unit exists; otherwise, <u>ex</u> is assigned the value false.

OPENED = <u>od</u>

<u>od</u> is a logical variable or logical array element. Execution of an INQUIRE by file statement causes <u>od</u> to be assigned the value true if the file specified is connected to a unit; otherwise, <u>od</u> is assigned the value false. Execution of an INQUIRE by unit statement causes <u>od</u> to be assigned the value true if the specified unit is connected to a file; otherwise, <u>od</u> is assigned the value false.

The INQUIRE statement is not included in the subset.

NUMBER = num

num is an integer variable or integer array element that is assigned the value of the external unit identifier of the unit that is currently connected to the file. If there is no unit connected to the file, num becomes undefined.

NAMED = nmd

nmd is a logical variable or logical array element that is assigned the value true if the file has a name; otherwise, it is assigned the value false.

NAME = fn

fn is a character variable or character array element that is assigned the value of the name of the file, if the file has a name; otherwise, it becomes undefined. Note that if this specifier appears in an INQUIRE by file statement, its value is not necessarily the same as the name given in the FILE= specifier. For example, the processor may return a file name qualified by a user identification. However, the value returned must be suitable for use as the value of a FILE= specifier in an OPEN statement.

ACCESS = acc

acc is a character variable or character array element that is assigned the value SEQUENTIAL if the file is connected for sequential access, and DIRECT if the file is connected for direct access. If there is no connection, acc becomes undefined.

SEQUENTIAL = seq

seq is a character variable or character array element that is assigned the value YES if SEQUENTIAL is included in the set of allowed access methods for the file, NO if SEQUENTIAL is not included in the set of allowed access methods for the file, and UNKNOWN if the processor is unable to determine whether or not SEQUENTIAL is included in the set of allowed access methods for the file.

DIRECT = dir

dir is a character variable or character array element that is assigned the value YES if DIRECT is included in the set of allowed access methods for the file, NO if DIRECT is not included in the set of allowed access methods for the file, and UNKNOWN if the processor is unable to determine whether or not

The INQUIRE statement is not included in the subset.

DIRECT is included in the set of allowed access methods for the file.

FORM = fm

fm is a character variable or character array element that is assigned the value FORMATTED if the file is connected for formatted input/output, and is assigned the value UNFORMATTED if the file is connected for unformatted input/output. If there is no connection, fm becomes undefined.

FORMATTED = fmt

fmt is a character variable or character array element that is assigned the value YES if FORMATTED is included in the set of allowed forms for the file, NO if FORMATTED is not included in the set of allowed forms for the file, and UNKNOWN if the processor is unable to determine whether or not FORMATTED is included in the set of allowed forms for the file.

UNFORMATTED = unf

unf is a character variable or character array element that is assigned the value YES if UNFORMATTED is included in the set of allowed forms for the file, NO if UNFORMATTED is not included in the set of allowed forms for the file, and UNKNOWN if the processor is unable to determine whether or not UNFORMATTED is included in the set of allowed forms for the file.

RECL = rcl

rcl is an integer variable or integer array element that is assigned the value of the record length of the file connected for direct access. If the file is connected for formatted input/output, the length is the number of characters. If the file is connected for unformatted input/output, the length is measured in processor-dependent units. If there is no connection or if the connection is not for direct access, rcl becomes undefined.

NEXTREC = nr

nr is an integer variable or integer array element that is assigned the value n+1, where n is the record number of the last record read or written on the file connected for direct access. If the file is connected but no records have been read or written since the connection, nr is assigned the value 1. If the file is not connected for direct access or if the position of the file is indeterminate because of a previous error condition, nr becomes undefined.

The INQUIRE statement is not included in the subset.

12.10.4 <u>File Positioning Statements</u>. The forms of the file positioning statements are:

BACKSPACE <u>u</u>

ENDFILE <u>u</u>

REWIND <u>u</u>

where: <u>u</u> is an external unit identifier (12.3.3)

BLANK = <u>blnk</u>

<u>blnk</u> is a character variable or character array element that is assigned the value NULL if null blank control is in effect for the file connected for formatted input/output, and is assigned the value ZERO if zero blank control is in effect for the file connected for formatted input/output. If there is no connection, or if the connection is not for formatted input/output, <u>blnk</u> becomes undefined.

A variable or array element that may become defined or undefined as a result of its use as a specifier in an INQUIRE statement, or any associated entity, must not be referenced by any other specifier in the same INQUIRE statement.

Execution of an INQUIRE by file statement causes the specifier variables or array elements <u>nmd</u>, <u>fn</u>, <u>seq</u>, <u>dir</u>, <u>fmt</u>, and <u>unf</u> to be assigned values only if the value of <u>fin</u> is acceptable to the processor as a file name and if there exists a file by that name; otherwise, they become undefined. Note that <u>num</u> becomes defined if and only if <u>od</u> becomes defined with the value true. Note also that the specifier variables or array elements <u>acc</u>, <u>fm</u>, <u>rcl</u>, <u>nr</u>, and <u>blnk</u> may become defined only if <u>od</u> becomes defined with the value true.

Execution of an INQUIRE by unit statement causes the specifier variables or array elements <u>num</u>, <u>nmd</u>, <u>fn</u>, <u>acc</u>, <u>seq</u>, <u>dir</u>, <u>fm</u>, <u>fmt</u>, <u>unf</u>, <u>rcl</u>, <u>nr</u>, and <u>blnk</u> to be assigned values only if the specified unit exists and if a file is connected to the unit; otherwise, they become undefined.

If an error condition occurs during execution of an INQUIRE statement, all of the inquiry specifier variables and array elements except <u>ios</u> become undefined.

Note that the specifier variables or array elements <u>ex</u> and <u>od</u> always become defined unless an error condition occurs.

12.10.4 <u>File Positioning Statements</u>. The forms of the file positioning statements are:

BACKSPACE <u>u</u>
BACKSPACE (<u>alist</u>)

ENDFILE <u>u</u>
ENDFILE (<u>alist</u>)

REWIND <u>u</u>
REWIND (<u>alist</u>)

where: <u>u</u> is an external unit identifier (12.3.3)

<u>alist</u> is a list (2.10) of specifiers:

The external unit specified by a BACKSPACE, ENDFILE, or REWIND statement must be connected for sequential access.

12.10.4.1 BACKSPACE Statement. Execution of a BACKSPACE statement causes the file connected to the specified unit to be positioned before the preceding record. If there is no preceding record, the position of the file is not changed. Note that if the preceding record is an endfile record, the file becomes positioned before the endfile record.

Backspacing a file that is connected but does not exist is prohibited.

12.10.4.2 ENDFILE Statement. Execution of an ENDFILE statement writes an endfile record as the next record of the file. The file is then positioned after the endfile record. If the file may also be connected for direct access, only those records before the endfile record are considered to have been written. Thus, only those records may be read during subsequent direct access connections to the file.

After execution of an ENDFILE statement, a BACKSPACE or REWIND statement must be used to reposition the file prior to execution of any data transfer input/output statement.

Execution of an ENDFILE statement for a file that is connected but does not exist creates the file.

12.10.4.3 REWIND Statement. Execution of a REWIND statement causes the specified file to be positioned at its initial point. Note that if the file is already positioned at its initial point, execution of this statement has no effect on the position of the file.

Execution of a REWIND statement for a file that is connected but does not exist is permitted but has no effect.

```
[UNIT =] u
IOSTAT = ios
ERR = s
```

alist must contain exactly one external unit specifier (12.3.3) and may contain at most one of each of the other specifiers.

The external unit specified by a BACKSPACE, ENDFILE, or REWIND statement must be connected for sequential access.

Execution of a file positioning statement containing an input/output status specifier causes *ios* to become defined with a zero value if no error condition exists or with a processor-dependent positive integer value if an error condition exists.

12.10.4.1 BACKSPACE Statement. Execution of a BACKSPACE statement causes the file connected to the specified unit to be positioned before the preceding record. If there is no preceding record, the position of the file is not changed. Note that if the preceding record is an endfile record, the file becomes positioned before the endfile record.

Backspacing a file that is connected but does not exist is prohibited.

Backspacing over records written using list-directed formatting is prohibited.

12.10.4.2 ENDFILE Statement. Execution of an ENDFILE statement writes an endfile record as the next record of the file. The file is then positioned after the endfile record. If the file may also be connected for direct access, only those records before the endfile record are considered to have been written. Thus, only those records may be read during subsequent direct access connections to the file.

After execution of an ENDFILE statement, a BACKSPACE or REWIND statement must be used to reposition the file prior to execution of any data transfer input/output statement.

Execution of an ENDFILE statement for a file that is connected but does not exist creates the file.

12.10.4.3 REWIND Statement. Execution of a REWIND statement causes the specified file to be positioned at its initial point. Note that if the file is already positioned at its initial point, execution of this statement has no effect on the position of the file.

Execution of a REWIND statement for a file that is connected but does not exist is permitted but has no effect.

12.11 Restrictions on Function References and List Items

Function references in input/output statements are not included in the subset.

12.12 Restriction on Input/Output Statements

If a unit, or a file connected to a unit, does not have all of the properties required for the execution of certain input/output statements, those statements must not refer to the unit.

12.11 Restrictions on Function References and List Items

A function must not be referenced within an expression appearing anywhere in an input/output statement if such a reference causes an input/output statement to be executed. Note that a restriction in the evaluation of expressions (6.6) prohibits certain side effects.

12.12 Restriction on Input/Output Statements

If a unit, or a file connected to a unit, does not have all of the properties required for the execution of certain input/output statements, those statements must not refer to the unit.

13. FORMAT SPECIFICATION

A format used in conjunction with formatted input/output statements provides information that directs the editing between the internal representation and the character strings of a record or a sequence of records in the file.

A format specification provides explicit editing information.

13.1 Format Specification Methods

Format specifications may be given:

(1) In FORMAT statements

(2) As character constants

13.1.1 FORMAT Statement. The form of a FORMAT statement is:

FORMAT *fs*

where *fs* is a format specification, as described in 13.2. The statement must be labeled.

13.1.2 Character Format Specification. The format identifier (12.4) in a formatted input/output statement may be a character constant if the leftmost character positions of the specified constant constitute a format specification.

A character format specification must be of the form described in 13.2. Note that the form begins with a left parenthesis and ends with a right parenthesis. Character data may follow the right parenthesis that ends the format specification, with no effect on the format specification. Blank characters may precede the format specification.

13. FORMAT SPECIFICATION

A format used in conjunction with formatted input/output statements provides information that directs the editing between the internal representation and the character strings of a record or a sequence of records in the file.

A format specification provides explicit editing information. An asterisk (*) as a format identifier in an input/output statement indicates list-directed formatting (13.6).

13.1 Format Specification Methods

Format specifications may be given:

(1) In FORMAT statements

(2) As values of character arrays, character variables, or other character expressions

13.1.1 <u>FORMAT Statement</u>. The form of a FORMAT statement is:

FORMAT <u>fs</u>

where <u>fs</u> is a format specification, as described in 13.2. The statement must be labeled.

13.1.2 <u>Character Format Specification</u>. If the format identifier (12.4) in a formatted input/output statement is a character array name, character variable name, or other character expression, the leftmost character positions of the specified entity must be in a defined state with character data that constitute a format specification when the statement is executed.

A character format specification must be of the form described in 13.2. Note that the form begins with a left parenthesis and ends with a right parenthesis. Character data may follow the right parenthesis that ends the format specification, with no effect on the format specification. Blank characters may precede the format specification.

If the format identifier is a character array name, the length of the format specification may exceed the length of the first element of the array; a character array format specification is considered to be a concatenation of all the array elements of the array in the order given by array element ordering (5.2.4). However, if a character array element name is specified as a format identifier, the length of the format specification must not exceed the length of the array element.

13.2 Form of a Format Specification

The form of a format specification is:

([flist])

where flist is a list (2.10). The forms of the flist items are:

[r] ed

ned

[r] fs

where: ed is a repeatable edit descriptor (13.2.1)

ned is a nonrepeatable edit descriptor (13.2.1)

fs is a format specification with a nonempty list flist

r is a nonzero, unsigned, integer constant called a repeat specification

The comma used to separate list items in the list flist may be omitted as follows:

(1) Between a P edit descriptor and an immediately following F or E edit descriptor (13.5.9)

(2) Before or after a slash edit descriptor (13.5.4)

At most three levels of parenthesis nesting are permitted within the outermost parentheses.

13.2.1 Edit Descriptors.

An edit descriptor is either a repeatable edit descriptor or a nonrepeatable edit descriptor.

The forms of a repeatable edit descriptor are:

Iw

Fw.d
Ew.d
Ew.dEe

Lw
A
Aw

where: I, F, E, L, and A indicate the manner of editing

13.2 Form of a Format Specification

The form of a *format specification* is:

([*flist*])

where *flist* is a list (2.10). The forms of the *flist* items are:

[*r*] *ed*

ned

[*r*] *fs*

where: *ed* is a repeatable edit descriptor (13.2.1)

ned is a nonrepeatable edit descriptor (13.2.1)

fs is a format specification with a nonempty list *flist*

r is a nonzero, unsigned, integer constant called a *repeat specification*

The comma used to separate list items in the list *flist* may be omitted as follows:

(1) Between a P edit descriptor and an immediately following F, E, D, or G edit descriptor (13.5.9)

(2) Before or after a slash edit descriptor (13.5.4)

(3) Before or after a colon edit descriptor (13.5.5)

13.2.1 Edit Descriptors.

An *edit descriptor* is either a repeatable edit descriptor or a nonrepeatable edit descriptor.

The forms of a *repeatable edit descriptor* are:

I*w*
I*w*.*m*
F*w*.*d*
E*w*.*d*
E*w*.*d*E*e*
D*w*.*d*
G*w*.*d*
G*w*.*d*E*e*
L*w*
A
A*w*

where: I, F, E, D, G, L, and A indicate the manner of editing *w* and *e* are nonzero, unsigned, integer constants

d is an unsigned integer constant

The forms of a nonrepeatable edit descriptor are:

'h_1 h_2 ... h_n'
nH$h_1 h_2$ ... h_n

nX
/

kP
BN
BZ

where apostrophe, H, X, slash, P, BN, and BZ indicate the manner of editing

h is one of the characters capable of representation by the processor

n is a nonzero, unsigned, integer constant

k is an optionally signed integer constant

13.3 Interaction Between Input/Output List and Format

The beginning of formatted data transfer using a format specification (12.9.5.2.1) initiates format control. Each action of format control depends on information jointly provided by:

(1) the next edit descriptor contained in the format specification, and

(2) the next item in the input/output list, if one exists.

If an input/output list specifies at least one list item, at least one repeatable edit descriptor must exist in the format specification. Note that an empty format specification of the form ( ) may be used only if no list items are specified; in this case, one input record is skipped or one output record containing no characters is written. Except for an edit descriptor preceded by a repeat specification, *r ed*, and a format specification preceded by a repeat specification, *r*(*flist*), a format specification is interpreted from left to right. A format specification or edit descriptor preceded by a repeat specification *r* is processed as a list of *r* format specifications or edit *w* and *e* are nonzero, unsigned, integer constants

d and *m* are unsigned integer constants

The forms of a *nonrepeatable edit descriptor* are:

'h_1 h_2 ... h_n'
*n*H$h_1$$h_2$ ... h_n
T*c*
TL*c*
TR*c*
*n*X
/
:
S
SP
SS
*k*P
BN
BZ

where: apostrophe, H, T, TL, TR, X, slash, colon, S, SP, SS, P, BN, and BZ indicate the manner of editing

h is one of the characters capable of representation by the processor

n and *c* are nonzero, unsigned, integer constants

k is an optionally signed integer constant

13.3 Interaction Between Input/Output List and Format

The beginning of formatted data transfer using a format specification (12.9.5.2.1) initiates *format control*. Each action of format control depends on information jointly provided by:

(1) the next edit descriptor contained in the format specification, and

(2) the next item in the input/output list, if one exists.

If an input/output list specifies at least one list item, at least one repeatable edit descriptor must exist in the format specification. Note that an empty format specification of the form ( ) may be used only if no list items are specified; in this case, one input record is skipped or one output record containing no characters is written. Except for an edit descriptor preceded by a repeat specification, *r ed*, and a format specification preceded by a repeat specification, *r*(*flist*), a format specification is interpreted from left to right. A format specification or edit descriptor preceded by a repeat specification *r* is processed as a list of *r* format specifications or edit descriptors identical to the format specification or edit descriptor without the repeat specification. Note that an omitted repeat specification is treated the same as a repeat specification whose value is one.

To each repeatable edit descriptor interpreted in a format specification, there corresponds one item specified by the input/output list (12.8.2). To each P, X, H, BN, BZ, slash, or apostrophe edit descriptor, there is no corresponding item specified by the input/output list, and format control communicates information directly with the record.

Whenever format control encounters a repeatable edit descriptor in a format specification, it determines whether there is a corresponding item specified by the input/output list. If there is such an item, it transmits appropriately edited information between the item and the record, and then format control proceeds. If there is no corresponding item, format control terminates.

If format control encounters the rightmost parenthesis of a complete format specification and another list item is not specified, format control terminates. However, if another list item is specified, the file is positioned at the beginning of the next record and format control then reverts to the beginning of the format specification terminated by the last preceding right parenthesis. If there is no such preceding right parenthesis, format control reverts to the first left parenthesis of the format specification. If such reversion occurs, the reused portion of the format specification must contain at least one repeatable edit descriptor. If format control reverts to a parenthesis that is preceded by a repeat specification, the repeat specification is reused. Reversion of format control, of itself, has no effect on the scale factor (13.5.7) or the BN or BZ edit descriptor blank control (13.5.8).

13.4 Positioning by Format Control

After each I, F, E, L, A, H, or apostrophe edit descriptor is processed, the file is positioned after the last character read or written in the current record.

After each X or slash edit descriptor is processed, the file is positioned as described in 13.5.3 and 13.5.4.

descriptors identical to the format specification or edit descriptor without the repeat specification. Note that an omitted repeat specification is treated the same as a repeat specification whose value is one.

To each repeatable edit descriptor interpreted in a format specification, there corresponds one item specified by the input/output list (12.8.2), except that a list item of type complex requires the interpretation of two F, E, D, or G edit descriptors. To each P, X, T, TL, TR, S, SP, SS, H, BN, BZ, slash, colon, or apostrophe edit descriptor, there is no corresponding item specified by the input/output list, and format control communicates information directly with the record.

Whenever format control encounters a repeatable edit descriptor in a format specification, it determines whether there is a corresponding item specified by the input/output list. If there is such an item, it transmits appropriately edited information between the item and the record, and then format control proceeds. If there is no corresponding item, format control terminates.

If format control encounters a colon edit descriptor in a format specification and another list item is not specified, format control terminates.

If format control encounters the rightmost parenthesis of a complete format specification and another list item is not specified, format control terminates. However, if another list item is specified, the file is positioned at the beginning of the next record and format control then reverts to the beginning of the format specification terminated by the last preceding right parenthesis. If there is no such preceding right parenthesis, format control reverts to the first left parenthesis of the format specification. If such reversion occurs, the reused portion of the format specification must contain at least one repeatable edit descriptor. If format control reverts to a parenthesis that is preceded by a repeat specification, the repeat specification is reused. Reversion of format control, of itself, has no effect on the scale factor (13.5.7), the S, SP, or SS edit descriptor sign control (13.5.6), or the BN or BZ edit descriptor blank control (13.5.8).

13.4 Positioning by Format Control

After each I, F, E, D, G, L, A, H, or apostrophe edit descriptor is processed, the file is positioned after the last character read or written in the current record.

After each T, TL, TR, X, or slash edit descriptor is processed, the file is positioned as described in 13.5.3 and 13.5.4.

If format control reverts as described in 13.3, the file is positioned in a manner identical to the way it is positioned when a slash edit descriptor is processed (13.5.4).

During a read operation, any unprocessed characters of the record are skipped whenever the next record is read.

13.5 Editing

Edit descriptors are used to specify the form of a record and to direct the editing between the characters in a record and internal representations of data.

A field is a part of a record that is read on input or written on output when format control processes one I, F, E, L, A, H, or apostrophe edit descriptor. The field width is the size in characters of the field.

The internal representation of a datum corresponds to the internal representation of a constant of the corresponding type (Section 4).

13.5.1 Apostrophe Editing. The apostrophe edit descriptor has the form of a character constant. It causes characters to be written from the enclosed characters (including blanks) of the edit descriptor itself. An apostrophe edit descriptor must not be used on input.

The width of the field is the number of characters contained in, but not including, the delimiting apostrophes. Within the field, two consecutive apostrophes with no intervening blanks are counted as a single apostrophe.

13.5.2 H Editing. The nH edit descriptor causes character information to be written from the n characters (including blanks) following the H of the nH edit descriptor in the format specification itself. An H edit descriptor must not be used on input.

Note that if an H edit descriptor occurs within a character constant and includes an apostrophe, the apostrophe must be represented by two consecutive apostrophes, which are counted as one character in specifying n.

13.5.3 Positional Editing. The X edit descriptor specifies the position at which the next character will be transmitted to or from the record.

The position specified by an X edit descriptor is forward from the current position. On input, a position beyond the

If format control reverts as described in 13.3, the file is positioned in a manner identical to the way it is positioned when a slash edit descriptor is processed (13.5.4).

During a read operation, any unprocessed characters of the record are skipped whenever the next record is read.

13.5 Editing

Edit descriptors are used to specify the form of a record and to direct the editing between the characters in a record and internal representations of data.

A *field* is a part of a record that is read on input or written on output when format control processes one I, F, E, D, G, L, A, H, or apostrophe edit descriptor. The *field width* is the size in characters of the field.

The internal representation of a datum corresponds to the internal representation of a constant of the corresponding type (Section 4).

13.5.1 Apostrophe Editing. The apostrophe edit descriptor has the form of a character constant. It causes characters to be written from the enclosed characters (including blanks) of the edit descriptor itself. An apostrophe edit descriptor must not be used on input.

The width of the field is the number of characters contained in, but not including, the delimiting apostrophes. Within the field, two consecutive apostrophes with no intervening blanks are counted as a single apostrophe.

13.5.2 H Editing. The *n*H edit descriptor causes character information to be written from the *n* characters (including blanks) following the H of the *n*H edit descriptor in the format specification itself. An H edit descriptor must not be used on input.

Note that if an H edit descriptor occurs within a character constant and includes an apostrophe, the apostrophe must be represented by two consecutive apostrophes, which are counted as one character in specifying *n*.

13.5.3 Positional Editing. The T, TL, TR, and X edit descriptors specify the position at which the next character will be transmitted to or from the record.

The position specified by a T edit descriptor may be in either direction from the current position. On input, this allows portions of a record to be processed more than once, possibly with different editing.

The position specified by an X edit descriptor is forward from the current position. On input, a position beyond the last character of the record may be specified if no characters are transmitted from such positions.

On output, an X edit descriptor does not by itself cause characters to be transmitted and therefore does not by itself affect the length of the record. If characters are transmitted to positions at or after the position specified by an X edit descriptor, positions skipped are filled with blanks. The result is as if the entire record were initially filled with blanks.

13.5.3.1 T, TL, and TR Editing. The T, TL, and TR edit descriptors are not included in the subset.

13.5.3.2 X Editing. The *n*X edit descriptor indicates that the transmission of the next character to or from a record is to occur at the position *n* characters forward from the current position.

13.5.4 Slash Editing. The slash edit descriptor indicates the end of data transfer on the current record.

On input from a file connected for sequential access, the remaining portion of the current record is skipped and the file is positioned at the beginning of the next record. This record becomes the current record. On output to a file connected for sequential access, a new record is created and becomes the last and current record of the file.

Note that a record that contains no characters may be written on output. If the file is an internal file or a file connected for direct access, the record is filled with blank characters. Note also that an entire record may be skipped on input.

last character of the record may be specified if no characters are transmitted from such positions.

On output, a T, TL, TR, or X edit descriptor does not by itself cause characters to be transmitted and therefore does not by itself affect the length of the record. If characters are transmitted to positions at or after the position specified by a T, TL, TR, or X edit descriptor, positions skipped and not previously filled are filled with blanks. The result is as if the entire record were initially filled with blanks.

On output, a character in the record may be replaced. However, a T, TL, TR, or X edit descriptor never directly causes a character already placed in the record to be replaced. Such edit descriptors may result in positioning so that subsequent editing causes a replacement.

13.5.3.1 <u>T, TL, and TR Editing</u>. The T*c* edit descriptor indicates that the transmission of the next character to or from a record is to occur at the *c*th character position.

The TL*c* edit descriptor indicates that the transmission of the next character to or from the record is to occur at the character position *c* characters backward from the current position. However, if the current position is less than or equal to position *c*, the TL*c* edit descriptor indicates that the transmission of the next character to or from the record is to occur at position one of the current record.

The TR*c* edit descriptor indicates that the transmission of the next character to or from the record is to occur at the character position *c* characters forward from the current position.

13.5.3.2 <u>X Editing</u>. The *n*X edit descriptor indicates that the transmission of the next character to or from a record is to occur at the position *n* characters forward from the current position.

13.5.4 <u>Slash Editing</u>. The slash edit descriptor indicates the end of data transfer on the current record.

On input from a file connected for sequential access, the remaining portion of the current record is skipped and the file is positioned at the beginning of the next record. This record becomes the current record. On output to a file connected for sequential access, a new record is created and becomes the last and current record of the file.

Note that a record that contains no characters may be written on output. If the file is an internal file or a file connected for direct access, the record is filled with blank characters. Note also that an entire record may be skipped on input.

13.5.5 Colon Editing. The colon edit descriptor is not included in the subset.

13.5.6 S, SP, and SS Editing. The S, SP, and SS edit descriptors are not included in the subset.

13.5.7 P Editing. A scale factor is specified by a P edit descriptor, which is of the form:

*k*P

where *k* is an optionally signed integer constant, called the *scale factor*.

13.5.7.1 Scale Factor. The value of the scale factor is zero at the beginning of execution of each input/output statement. It applies to all subsequently interpreted F and E edit descriptors until another scale factor is encountered, and then that scale factor is established. Note that reversion of format control (13.3) does not affect the established scale factor.

The scale factor *k* affects the appropriate editing in the following manner:

(1) On input, with F and E editing (provided that no exponent exists in the field) and F output editing, the scale factor effect is that the externally represented number equals the internally represented number multiplied by 10***k*.

(2) On input, with F and E editing, the scale factor has no effect if there is an exponent in the field.

For a file connected for direct access, the record number is increased by one and the file is positioned at the beginning of the record that has that record number. This record becomes the current record.

13.5.5 Colon Editing. The colon edit descriptor terminates format control if there are no more items in the input/output list (13.3). The colon edit descriptor has no effect if there are more items in the input/output list.

13.5.6 S, SP, and SS Editing. The S, SP, and SS edit descriptors may be used to control optional plus characters in numeric output fields. At the beginning of execution of each formatted output statement, the processor has the option of producing a plus in numeric output fields. If an SP edit descriptor is encountered in a format specification, the processor must produce a plus in any subsequent position that normally contains an optional plus. If an SS edit descriptor is encountered, the processor must not produce a plus in any subsequent position that normally contains an optional plus. If an S edit descriptor is encountered, the option of producing the plus is restored to the processor.

The S, SP, and SS edit descriptors affect only I, F, E, D, and G editing during the execution of an output statement. The S, SP, and SS edit descriptors have no effect during the execution of an input statement.

13.5.7 P Editing. A scale factor is specified by a P edit descriptor, which is of the form:

    kP

where k is an optionally signed integer constant, called the scale factor.

13.5.7.1 Scale Factor. The value of the scale factor is zero at the beginning of execution of each input/output statement. It applies to all subsequently interpreted F, E, D, and G edit descriptors until another scale factor is encountered, and then that scale factor is established. Note that reversion of format control (13.3) does not affect the established scale factor.

The scale factor k affects the appropriate editing in the following manner:

(1) On input, with F, E, D, and G editing (provided that no exponent exists in the field) and F output editing, the scale factor effect is that the externally represented number equals the internally represented number multiplied by 10**k.

(2) On input, with F, E, D, and G editing, the scale factor has no effect if there is an exponent in the field.

(3) On output, with E editing, the basic real constant (4.4.1) part of the quantity to be produced is multiplied by 10**k and the exponent is reduced by k.

13.5.8 BN and BZ Editing. The BN and BZ edit descriptors may be used to specify the interpretation of blanks, other than leading blanks, in numeric input fields. At the beginning of execution of each formatted input statement, such blank characters are interpreted as zeros. If a BN edit descriptor is encountered in a format specification, all such blank characters in succeeding numeric input fields are ignored. The effect of ignoring blanks is to treat the input field as if blanks had been removed, the remaining portion of the field right-justified, and the blanks replaced as leading blanks. However, a field of all blanks has the value zero. If a BZ edit descriptor is encountered in a format specification, all such blank characters in succeeding numeric input fields are treated as zeros.

The BN and BZ edit descriptors affect only I, F, and E editing during execution of an input statement. They have no effect during execution of an output statement.

13.5.9 Numeric Editing. The I, F, and E edit descriptors are used to specify input/output of integer and real data. The following general rules apply:

(1) On input, leading blanks are not significant. The interpretation of blanks, other than leading blanks, is determined by any BN or BZ blank control that is currently in effect for the unit (13.5.8). Plus signs may be omitted. A field of all blanks is considered to be zero.

(2) On input, with F and E editing, a decimal point appearing in the input field overrides the portion of an edit descriptor that specifies the decimal point location. The input field may have more digits than the processor uses to approximate the value of the datum.

(3) On output, the representation of a positive or zero internal value in the field may be prefixed with a plus, as controlled by the processor. The

(3) On output, with E and D editing, the basic real constant (4.4.1) part of the quantity to be produced is multiplied by 10**k and the exponent is reduced by k.

(4) On output, with G editing, the effect of the scale factor is suspended unless the magnitude of the datum to be edited is outside the range that permits the use of F editing. If the use of E editing is required, the scale factor has the same effect as with E output editing.

13.5.8 BN and BZ Editing. The BN and BZ edit descriptors may be used to specify the interpretation of blanks, other than leading blanks, in numeric input fields. At the beginning of execution of each formatted input statement, such blank characters are interpreted as zeros or are ignored, depending on the value of the BLANK= specifier (12.10.1) currently in effect for the unit. If a BN edit descriptor is encountered in a format specification, all such blank characters in succeeding numeric input fields are ignored. The effect of ignoring blanks is to treat the input field as if blanks had been removed, the remaining portion of the field right-justified, and the blanks replaced as leading blanks. However, a field of all blanks has the value zero. If a BZ edit descriptor is encountered in a format specification, all such blank characters in succeeding numeric input fields are treated as zeros.

The BN and BZ edit descriptors affect only I, F, E, D, and G editing during execution of an input statement. They have no effect during execution of an output statement.

13.5.9 Numeric Editing. The I, F, E, D, and G edit descriptors are used to specify input/output of integer, real, double precision, and complex data. The following general rules apply:

(1) On input, leading blanks are not significant. The interpretation of blanks, other than leading blanks, is determined by a combination of any BLANK= specifier and any BN or BZ blank control that is currently in effect for the unit (13.5.8). Plus signs may be omitted. A field of all blanks is considered to be zero.

(2) On input, with F, E, D, and G editing, a decimal point appearing in the input field overrides the portion of an edit descriptor that specifies the decimal point location. The input field may have more digits than the processor uses to approximate the value of the datum.

(3) On output, the representation of a positive or zero internal value in the field may be prefixed with a plus, as controlled by the S, SP, and SS edit

representation of a negative internal value in the field must be prefixed with a minus. However, the processor must not produce a negative signed zero in a formatted output record.

(4) On output, the representation is right-justified in the field. If the number of characters produced by the editing is smaller than the field width, leading blanks will be inserted in the field.

(5) On output, if the number of characters produced exceeds the field width or if an exponent exceeds its specified length using the E<u>w</u>.<u>d</u>E<u>e</u> edit descriptor, the processor will fill the entire field of width <u>w</u> with asterisks. However, the processor must not produce asterisks if the field width is not exceeded when optional characters are omitted.

13.5.9.1 <u>Integer Editing</u>. The I<u>w</u> edit descriptor indicates that the field to be edited occupies <u>w</u> positions. The specified input/output list item must be of type integer. On input, the specified list item will become defined with an integer datum. On output, the specified list item must be defined with an integer datum.

In the input field, the character string must be in the form of an optionally signed integer constant, except for the interpretation of blanks (13.5.9, item (1)).

The output field for the I<u>w</u> edit descriptor consists of zero or more leading blanks followed by a minus if the value of the internal datum is negative, or an optional plus otherwise, followed by the magnitude of the internal value in the form of an unsigned integer constant without leading zeros. Note that an integer constant always consists of at least one digit.

13.5.9.2 <u>Real and Double Precision Editing</u>. The F and E edit descriptors specify the editing of real data. An input/output list item corresponding to an F or E edit descriptor must be real. An input list item will become descriptors (13.5.6) or the processor. The representation of a negative internal value in the field must be prefixed with a minus. However, the processor must not produce a negative signed zero in a formatted output record.

(4) On output, the representation is right-justified in the field. If the number of characters produced by the editing is smaller than the field width, leading blanks will be inserted in the field.

(5) On output, if the number of characters produced exceeds the field width or if an exponent exceeds its specified length using the E*w*.*d*E*e* or G*w*.*d*E*e* edit descriptor, the processor will fill the entire field of width *w* with asterisks. However, the processor must not produce asterisks if the field width is not exceeded when optional characters are omitted. Note that when an SP edit descriptor is in effect, a plus is not optional (13.5.6).

13.5.9.1 Integer Editing. The I*w* and I*w*.*m* edit descriptors indicate that the field to be edited occupies *w* positions. The specified input/output list item must be of type integer. On input, the specified list item will become defined with an integer datum. On output, the specified list item must be defined with an integer datum.

On input, an I*w*.*m* edit descriptor is treated identically to an I*w* edit descriptor.

In the input field, the character string must be in the form of an optionally signed integer constant, except for the interpretation of blanks (13.5.9, item (1)).

The output field for the I*w* edit descriptor consists of zero or more leading blanks followed by a minus if the value of the internal datum is negative, or an optional plus otherwise, followed by the magnitude of the internal value in the form of an unsigned integer constant without leading zeros. Note that an integer constant always consists of at least one digit.

The output field for the I*w*.*m* edit descriptor is the same as for the I*w* edit descriptor, except that the unsigned integer constant consists of at least *m* digits and, if necessary, has leading zeros. The value of *m* must not exceed the value of *w*. If *m* is zero and the value of the internal datum is zero, the output field consists of only blank characters, regardless of the sign control in effect.

13.5.9.2 Real and Double Precision Editing. The F, E, D, and G edit descriptors specify the editing of real, double precision, and complex data. An input/output list item corresponding to an F, E, D, or G edit descriptor must be real, double precision, or complex. An input list item will defined with a real datum. An output list item must be defined with a real datum.

13.5.9.2.1 F Editing. The F*w*.*d* edit descriptor indicates that the field occupies *w* positions, the fractional part of which consists of *d* digits.

The input field consists of an optional sign, followed by a string of digits optionally containing a decimal point. If the decimal point is omitted, the rightmost *d* digits of the string, with leading zeros assumed if necessary, are interpreted as the fractional part of the value represented. The string of digits may contain more digits than a processor uses to approximate the value of the constant. The basic form may be followed by an exponent of one of the following forms:

(1) Signed integer constant

(2) E followed by zero or more blanks, followed by an optionally signed integer constant

(3) D followed by zero or more blanks, followed by an optionally signed integer constant

An exponent containing a D is processed identically to an exponent containing an E.

The output field consists of blanks, if necessary, followed by a minus if the internal value is negative, or an optional plus otherwise, followed by a string of digits that contains a decimal point and represents the magnitude of the internal value, as modified by the established scale factor and rounded to *d* fractional digits. Leading zeros are not permitted except for an optional zero immediately to the left of the decimal point if the magnitude of the value in the output field is less than one. The optional zero must appear if there would otherwise be no digits in the output field.

13.5.9.2.2 E and D Editing. The E*w*.*d* and E*w*.*d*E*e* edit descriptors indicate that the external field occupies *w* positions, the fractional part of which consists of *d* digits, unless a scale factor greater than one is in effect, and the exponent part consists of *e* digits. The *e* has no effect on input.

The form of the input field is the same as for F editing (13.5.9.2.1).

The form of the output field for a scale factor of zero is:

$$[\pm]\ [0]\ .\ x_1 x_2 \ldots x_d\ \underline{exp}$$

where: ± signifies a plus or a minus (13.5.9) become defined with a datum whose type is the same as that of the list item. An output list item must be defined with a datum whose type is the same as that of the list item.

13.5.9.2.1 F Editing. The F$\underline{w}$.$\underline{d}$ edit descriptor indicates that the field occupies $\underline{w}$ positions, the fractional part of which consists of $\underline{d}$ digits.

The input field consists of an optional sign, followed by a string of digits optionally containing a decimal point. If the decimal point is omitted, the rightmost $\underline{d}$ digits of the string, with leading zeros assumed if necessary, are interpreted as the fractional part of the value represented. The string of digits may contain more digits than a processor uses to approximate the value of the constant. The basic form may be followed by an exponent of one of the following forms:

(1) Signed integer constant

(2) E followed by zero or more blanks, followed by an optionally signed integer constant

(3) D followed by zero or more blanks, followed by an optionally signed integer constant

An exponent containing a D is processed identically to an exponent containing an E.

The output field consists of blanks, if necessary, followed by a minus if the internal value is negative, or an optional plus otherwise, followed by a string of digits that contains a decimal point and represents the magnitude of the internal value, as modified by the established scale factor and rounded to $\underline{d}$ fractional digits. Leading zeros are not permitted except for an optional zero immediately to the left of the decimal point if the magnitude of the value in the output field is less than one. The optional zero must appear if there would otherwise be no digits in the output field.

13.5.9.2.2 E and D Editing. The E$\underline{w}$.$\underline{d}$, D$\underline{w}$.$\underline{d}$, and E$\underline{w}$.$\underline{d}$E$\underline{e}$ edit descriptors indicate that the external field occupies $\underline{w}$ positions, the fractional part of which consists of $\underline{d}$ digits, unless a scale factor greater than one is in effect, and the exponent part consists of $\underline{e}$ digits. The $\underline{e}$ has no effect on input.

The form of the input field is the same as for F editing (13.5.9.2.1).

The form of the output field for a scale factor of zero is:

$[\pm]\ [0]\ .\ x_1 x_2 \ldots x_d\ \underline{exp}$

where: ± signifies a plus or a minus (13.5.9)

$x_1x_2 \ldots x_d$ are the $\underline{d}$ most significant digits of the value of the datum after rounding

$\underline{exp}$ is a decimal exponent, of one of the following forms:

Edit Descriptor	Absolute Value of Exponent	Form of Exponent
$E\underline{w}.\underline{d}$	$\lvert\underline{exp}\rvert \le 99$	$E\pm\underline{z}_1\underline{z}_2$ or $\pm 0\underline{z}_1\underline{z}_2$
	$99 < \lvert\underline{exp}\rvert \le 999$	$\pm\underline{z}_1\underline{z}_2\underline{z}_3$
$E\underline{w}.\underline{d}E\underline{e}$	$\lvert\underline{exp}\rvert \le (10^{**}\underline{e})-1$	$E\pm\underline{z}_1\underline{z}_2 \ldots \underline{z}_e$

where $\underline{z}$ is a digit. The sign in the exponent is required. A plus sign must be used if the exponent value is zero. The form $E\underline{w}.\underline{d}$ must not be used if $\lvert\underline{exp}\rvert > 999$.

The scale factor $\underline{k}$ controls the decimal normalization (13.5.7). If $-\underline{d} < \underline{k} \le 0$, the output field contains exactly $\lvert\underline{k}\rvert$ leading zeros and $\underline{d} - \lvert\underline{k}\rvert$ significant digits after the decimal point. If $0 < \underline{k} < \underline{d} + 2$, the output field contains exactly $\underline{k}$ significant digits to the left of the decimal point and $\underline{d} - \underline{k} + 1$ significant digits to the right of the decimal point. Other values of $\underline{k}$ are not permitted.

13.5.9.2.3 G Editing. The G edit descriptor is not included in the subset.

$x_1x_2...x_d$ are the *d* most significant digits of the value of the datum after rounding

exp is a decimal exponent, of one of the following forms:

Edit Descriptor	Absolute Value of Exponent	Form of Exponent
E*w*.*d*	$\lvert exp \rvert \le 99$	$E\pm z_1z_2$ or $\pm 0z_1z_2$
	$99 < \lvert exp \rvert \le 999$	$\pm z_1z_2z_3$
E*w*.*d*E*e*	$\lvert exp \rvert \le (10**e)-1$	$E\pm z_1z_2...z_e$
D*w*.*d*	$\lvert exp \rvert \le 99$	$D\pm z_1z_2$ or $E\pm z_1z_2$ or $\pm 0z_1z_2$
	$99 < \lvert exp \rvert \le 999$	$\pm z_1z_2z_3$

where *z* is a digit. The sign in the exponent is required. A plus sign must be used if the exponent value is zero. The forms E*w*.*d* and D*w*.*d* must not be used if $\lvert exp \rvert > 999$.

The scale factor *k* controls the decimal normalization (13.5.7). If $-d < k \le 0$, the output field contains exactly $\lvert k \rvert$ leading zeros and $d - \lvert k \rvert$ significant digits after the decimal point. If $0 < k < d + 2$, the output field contains exactly *k* significant digits to the left of the decimal point and $d - k + 1$ significant digits to the right of the decimal point. Other values of *k* are not permitted.

13.5.9.2.3 G Editing. The G*w*.*d* and G*w*.*d*E*e* edit descriptors indicate that the external field occupies *w* positions, the fractional part of which consists of *d* digits, unless a scale factor greater than one is in effect, and the exponent part consists of *e* digits.

G input editing is the same as for F editing (13.5.9.2.1).

The method of representation in the output field depends on the magnitude of the datum being edited. Let N be the magnitude of the internal datum. If $N < 0.1$ or $N \ge 10**d$, G*w*.*d* output editing is the same as *k*PE*w*.*d* output editing and G*w*.*d*E*e* output editing is the same as *k*PE*w*.*d*E*e* output editing, where *k* is the scale factor currently in effect. If N is greater than or equal to 0.1 and is less than $10**d$, the scale factor has no effect, and the value of N determines the editing as follows:

13.5.9.2.4 Complex Editing. Complex type is not included in the subset.

13.5.10 L Editing. The L*w* edit descriptor indicates that the field occupies *w* positions. The specified input/output list item must be of type logical. On input, the list item will become defined with a logical datum. On output, the specified list item must be defined with a logical datum.

The input field consists of optional blanks, optionally followed by a decimal point, followed by a T for true or F for false. The T or F may be followed by additional characters in the field. Note that the logical constants .TRUE. and .FALSE. are acceptable input forms.

The output field consists of *w* - 1 blanks followed by a T or F, as the value of the internal datum is true or false, respectively.

13.5.11 A Editing. The A[*w*] edit descriptor is used with an input/output list item of type character. On input, the input list item will become defined with character data. On output, the output list item must be defined with character data.

If a field width *w* is specified with the A edit descriptor, the field consists of *w* characters. If a field width *w* is

Magnitude of Datum	Equivalent Conversion
0.1≤N<1	F(w-n).d, n('b')
1≤N<10	F(w-n).(d-1), n('b')
.	.
.	.
.	.
10**(d-2)≤N<10**(d-1)	F(w-n).1, n('b')
10**(d-1)≤N<10**d	F(w-n).0, n('b')

where: b is a blank

n is 4 for Gw.d and e+2 for Gw.dEe

Note that the scale factor has no effect unless the magnitude of the datum to be edited is outside of the range that permits effective use of F editing.

13.5.9.2.4 Complex Editing. A complex datum consists of a pair of separate real data; therefore, the editing is specified by two successively interpreted F, E, D, or G edit descriptors. The first of the edit descriptors specifies the real part; the second specifies the imaginary part. The two edit descriptors may be different. Note that nonrepeatable edit descriptors may appear between the two successive F, E, D, or G edit descriptors.

13.5.10 L Editing. The Lw edit descriptor indicates that the field occupies w positions. The specified input/output list item must be of type logical. On input, the list item will become defined with a logical datum. On output, the specified list item must be defined with a logical datum.

The input field consists of optional blanks, optionally followed by a decimal point, followed by a T for true or F for false. The T or F may be followed by additional characters in the field. Note that the logical constants .TRUE. and .FALSE. are acceptable input forms.

The output field consists of w - 1 blanks followed by a T or F, as the value of the internal datum is true or false, respectively.

13.5.11 A Editing. The A[w] edit descriptor is used with an input/output list item of type character. On input, the input list item will become defined with character data. On output, the output list item must be defined with character data.

If a field width w is specified with the A edit descriptor, the field consists of w characters. If a field width w is not specified with the A edit descriptor, the number of characters in the field is the length of the character input/output list item.

Let _len_ be the length of the input/output list item. If the specified field width _w_ for A input is greater than or equal to _len_, the rightmost _len_ characters will be taken from the input field. If the specified field width is less than _len_, the _w_ characters will appear left-justified with _len_-_w_ trailing blanks in the internal representation.

If the specified field width _w_ for A output is greater than _len_, the output field will consist of _w_-_len_ blanks followed by the _len_ characters from the internal representation. If the specified field width _w_ is less than or equal to _len_, the output field will consist of the leftmost _w_ characters from the internal representation.

13.6 List-Directed Formatting

List-directed formatting is not included in the subset.

not specified with the A edit descriptor, the number of characters in the field is the length of the character input/output list item.

Let _len_ be the length of the input/output list item. If the specified field width _w_ for A input is greater than or equal to _len_, the rightmost _len_ characters will be taken from the input field. If the specified field width is less than _len_, the _w_ characters will appear left-justified with _len-w_ trailing blanks in the internal representation.

If the specified field width _w_ for A output is greater than _len_, the output field will consist of _w-len_ blanks followed by the _len_ characters from the internal representation. If the specified field width _w_ is less than or equal to _len_, the output field will consist of the leftmost _w_ characters from the internal representation.

13.6 List-Directed Formatting

The characters in one or more list-directed records constitute a sequence of values and value separators. The end of a record has the same effect as a blank character, unless it is within a character constant. Any sequence of two or more consecutive blanks is treated as a single blank, unless it is within a character constant.

Each value is either a constant, a null value, or of one of the forms:

_r*c_

_r*_

where _r_ is an unsigned, nonzero, integer constant. The _r*c_ form is equivalent to _r_ successive appearances of the constant _c_, and the _r*_ form is equivalent to _r_ successive null values. Neither of these forms may contain embedded blanks, except where permitted within the constant _c_.

A _value separator_ is one of the following:

(1) A comma optionally preceded by one or more contiguous blanks and optionally followed by one or more contiguous blanks

(2) A slash optionally preceded by one or more contiguous blanks and optionally followed by one or more contiguous blanks

(3) One or more contiguous blanks between two constants or following the last constant

13.6.1 List-Directed Input. List-directed formatting is not included in the subset.

13.6.1 <u>List-Directed Input</u>. Input forms acceptable to format specifications for a given type are acceptable for list-directed formatting, except as noted below. The form of the input value must be acceptable for the type of the input list item. Blanks are never used as zeros, and embedded blanks are not permitted in constants, except within character constants and complex constants as specified below. Note that the end of a record has the effect of a blank, except when it appears within a character constant.

When the corresponding input list item is of type real or double precision, the input form is that of a numeric input field. A <u>numeric input field</u> is a field suitable for F editing (13.5.9.2) that is assumed to have no fractional digits unless a decimal point appears within the field.

When the corresponding list item is of type complex, the input form consists of a left parenthesis followed by an ordered pair of numeric input fields separated by a comma, and followed by a right parenthesis. The first numeric input field is the real part of the complex constant and the second is the imaginary part. Each of the numeric input fields may be preceded or followed by blanks. The end of a record may occur between the real part and the comma or between the comma and the imaginary part.

When the corresponding list item is of type logical, the input form must not include either slashes or commas among the optional characters permitted for L editing (13.5.10).

When the corresponding list item is of type character, the input form consists of a nonempty string of characters enclosed in apostrophes. Each apostrophe within a character constant must be represented by two consecutive apostrophes without an intervening blank or end of record. Character constants may be continued from the end of one record to the beginning of the next record. The end of the record does not cause a blank or any other character to become part of the constant. The constant may be continued on as many records as needed. The characters blank, comma, and slash may appear in character constants.

Let <u>len</u> be the length of the list item, and let <u>w</u> be the length of the character constant. If <u>len</u> is less than or equal to <u>w</u>, the leftmost <u>len</u> characters of the constant are transmitted to the list item. If <u>len</u> is greater than <u>w</u>, the constant is transmitted to the leftmost <u>w</u> characters of the list item and the remaining <u>len-w</u> characters of the list item are filled with blanks. Note that the effect is as though the constant were assigned to the list item in a character assignment statement (10.4).

A null value is specified by having no characters between successive value separators, no characters preceding the first value separator in the first record read by each

13.6.2 List-Directed Output. List-directed formatting is not included in the subset.

execution of a list-directed input statement, or the *r** form. A null value has no effect on the definition status of the corresponding input list item. If the input list item is defined, it retains its previous value; if it is undefined, it remains undefined. A null value may not be used as either the real or imaginary part of a complex constant, but a single null value may represent an entire complex constant. Note that the end of a record following any other separator, with or without separating blanks, does not specify a null value.

A slash encountered as a value separator during execution of a list-directed input statement causes termination of execution of that input statement after the assignment of the previous value. If there are additional items in the input list, the effect is as if null values had been supplied for them.

Note that all blanks in a list-directed input record are considered to be part of some value separator except for the following:

(1) Blanks embedded in a character constant

(2) Embedded blanks surrounding the real or imaginary part of a complex constant

(3) Leading blanks in the first record read by each execution of a list-directed input statement, unless immediately followed by a slash or comma

13.6.2 List-Directed Output. The form of the values produced is the same as that required for input, except as noted otherwise. With the exception of character constants, the values are separated by one of the following:

(1) One or more blanks

(2) A comma optionally preceded by one or more blanks and optionally followed by one or more blanks

The processor may begin new records as necessary, but, except for complex constants and character constants, the end of a record must not occur within a constant and blanks must not appear within a constant.

Logical output constants are T for the value true and F for the value false.

Integer output constants are produced with the effect of an I*w* edit descriptor, for some reasonable value of *w*.

Real and double precision constants are produced with the effect of either an F edit descriptor or an E edit descriptor, depending on the magnitude *x* of the value and a range $10^{**}d_1 \le x < 10^{**}d_2$, where d_1 and d_2 are processor-

List-directed formatting is not included in the subset.

dependent integer values. If the magnitude _x_ is within this range, the constant is produced using 0PF_w_._d_; otherwise, 1PE_w_._d_E_e_ is used. Reasonable processor-dependent values of _w_, _d_, and _e_ are used for each of the cases involved.

Complex constants are enclosed in parentheses, with a comma separating the real and imaginary parts. The end of a record may occur between the comma and the imaginary part only if the entire constant is as long as, or longer than, an entire record. The only embedded blanks permitted within a complex constant are between the comma and the end of a record and one blank at the beginning of the next record.

Character constants produced are not delimited by apostrophes, are not preceded or followed by a value separator, have each internal apostrophe represented externally by one apostrophe, and have a blank character inserted by the processor for carriage control at the beginning of any record that begins with the continuation of a character constant from the preceding record.

If two or more successive values in an output record produced have identical values, the processor has the option of producing a repeated constant of the form _r_*_c_ instead of the sequence of identical values.

Slashes, as value separators, and null values are not produced by list-directed formatting.

Each output record begins with a blank character to provide carriage control when the record is printed.

14. MAIN PROGRAM

A <u>main program</u> is a program unit that does not have a FUNCTION or SUBROUTINE statement as its first statement. It may have a PROGRAM statement as its first statement.

There must be exactly one main program in an executable program. Execution of an executable program begins with the execution of the first executable statement of the main program.

14.1 PROGRAM Statement

The form of a PROGRAM statement is:

PROGRAM *pgm*

where *pgm* is the symbolic name of the main program in which the PROGRAM statement appears.

A PROGRAM statement is not required to appear in an executable program. If it does appear, it must be the first statement of the main program.

The symbolic name *pgm* is global (18.1.1) to the executable program and must not be the same as the name of an external procedure or common block in the same executable program. The name *pgm* must not be the same as any local name in the main program.

14.2 Main Program Restrictions

The PROGRAM statement may appear only as the first statement of a main program. A main program may contain any other statement except a FUNCTION, SUBROUTINE, or RETURN statement. The appearance of a SAVE statement in a main program has no effect.

A main program may not be referenced from a subprogram or from itself.

14. MAIN PROGRAM

A <u>main program</u> is a program unit that does not have a FUNCTION, SUBROUTINE, or BLOCK DATA statement as its first statement. It may have a PROGRAM statement as its first statement.

There must be exactly one main program in an executable program. Execution of an executable program begins with the execution of the first executable statement of the main program.

14.1 <u>PROGRAM Statement</u>

The form of a PROGRAM statement is:

PROGRAM <u>pgm</u>

where <u>pgm</u> is the symbolic name of the main program in which the PROGRAM statement appears.

A PROGRAM statement is not required to appear in an executable program. If it does appear, it must be the first statement of the main program.

The symbolic name <u>pgm</u> is global (18.1.1) to the executable program and must not be the same as the name of an external procedure, block data subprogram, or common block in the same executable program. The name <u>pgm</u> must not be the same as any local name in the main program.

14.2 <u>Main Program Restrictions</u>

The PROGRAM statement may appear only as the first statement of a main program. A main program may contain any other statement except a BLOCK DATA, FUNCTION, SUBROUTINE, ENTRY, or RETURN statement. The appearance of a SAVE statement in a main program has no effect.

A main program may not be referenced from a subprogram or from itself.

15. FUNCTIONS AND SUBROUTINES

15.1 Categories of Functions and Subroutines

15.1.1 Procedures. Functions and subroutines are procedures. There are four categories of procedures:

(1) Intrinsic functions

(2) Statement functions

(3) External functions

(4) Subroutines

Intrinsic functions, statement functions, and external functions are referred to collectively as functions.

External functions and subroutines are referred to collectively as external procedures.

15.1.2 External Functions. There are two categories of external functions:

(1) External functions specified in function subprograms

(2) External functions specified by means other than FORTRAN subprograms

15.1.3 Subroutines. There are two categories of subroutines:

(1) Subroutines specified in subroutine subprograms

(2) Subroutines specified by means other than FORTRAN subprograms

15.1.4 Dummy Procedure. A dummy procedure is a dummy argument that is identified as a procedure (18.2.11).

15.2 Referencing a Function

A function is referenced in an expression and supplies a value to the expression. The value supplied is the value of the function.

An intrinsic function may be referenced in the main program or in any procedure subprogram of an executable program.

A statement function may be referenced only in the program unit in which the statement function statement appears.

An external function specified by a function subprogram may be referenced within any other procedure subprogram or the main program of the executable program. A subprogram must not reference itself, either directly or indirectly.

15. FUNCTIONS AND SUBROUTINES

15.1 Categories of Functions and Subroutines

15.1.1 Procedures. Functions and subroutines are procedures. There are four categories of procedures:

(1) Intrinsic functions

(2) Statement functions

(3) External functions

(4) Subroutines

Intrinsic functions, statement functions, and external functions are referred to collectively as functions.

External functions and subroutines are referred to collectively as external procedures.

15.1.2 External Functions. There are two categories of external functions:

(1) External functions specified in function subprograms

(2) External functions specified by means other than FORTRAN subprograms

15.1.3 Subroutines. There are two categories of subroutines:

(1) Subroutines specified in subroutine subprograms

(2) Subroutines specified by means other than FORTRAN subprograms

15.1.4 Dummy Procedure. A dummy procedure is a dummy argument that is identified as a procedure (18.2.11).

15.2 Referencing a Function

A function is referenced in an expression and supplies a value to the expression. The value supplied is the value of the function.

An intrinsic function may be referenced in the main program or in any procedure subprogram of an executable program.

A statement function may be referenced only in the program unit in which the statement function statement appears.

An external function specified by a function subprogram may be referenced within any other procedure subprogram or the main program of the executable program. A subprogram must not reference itself, either directly or indirectly.

An external function specified by means other than a subprogram may be referenced within any procedure subprogram or the main program of the executable program.

15.2.1 Form of a Function Reference. A function reference is used to reference an intrinsic function, statement function, or external function.

The form of a function reference is:

fun ([a [,a]...])

where: fun is the symbolic name of a function or a dummy procedure

a is an actual argument

The type of the result of a statement function or external function reference is the same as the type of the function name. The type is specified in the same manner as for variables and arrays (4.1.2). The type of the result of an intrinsic function is specified in Table 5 (15.10). A function must not be of type character.

15.2.2 Execution of a Function Reference. A function reference may appear only as a primary in an arithmetic or logical expression. Execution of a function reference in an expression causes the evaluation of the function identified by fun.

Return of control from a referenced function completes execution of the function reference. The value of the function is available to the referencing expression.

15.3 Intrinsic Functions

Intrinsic functions are supplied by the processor and have a special meaning. The specific names that identify the intrinsic functions, their function definitions, type of arguments, and type of results appear in Table 5.

An IMPLICIT statement does not change the type of an intrinsic function.

15.3.1 Specific Names and Generic Names. Only a specific intrinsic function name may be used as an actual argument when the argument is an intrinsic function.

An external function specified by means other than a subprogram may be referenced within any procedure subprogram or the main program of the executable program.

If a character function is referenced in a program unit, the function length specified in the program unit must be an integer constant expression.

15.2.1 Form of a Function Reference. A function reference is used to reference an intrinsic function, statement function, or external function.

The form of a function reference is:

    fun ([a [,a]...])

where: fun is the symbolic name of a function or a dummy procedure

a is an actual argument

The type of the result of a statement function or external function reference is the same as the type of the function name. The type is specified in the same manner as for variables and arrays (4.1.2). The type of the result of an intrinsic function is specified in Table 5 (15.10).

15.2.2 Execution of a Function Reference. A function reference may appear only as a primary in an arithmetic, logical, or character expression. Execution of a function reference in an expression causes the evaluation of the function identified by fun.

Return of control from a referenced function completes execution of the function reference. The value of the function is available to the referencing expression.

15.3 Intrinsic Functions

Intrinsic functions are supplied by the processor and have a special meaning. The specific names that identify the intrinsic functions, their generic names, function definitions, type of arguments, and type of results appear in Table 5.

An IMPLICIT statement does not change the type of an intrinsic function.

15.3.1 Specific Names and Generic Names. Generic names simplify the referencing of intrinsic functions, because the same function name may be used with more than one type of argument. Only a specific intrinsic function name may be used as an actual argument when the argument is an intrinsic function.

For those intrinsic functions that have more than one argument, all arguments must be of the same type.

If the specific name of an intrinsic function appears in the dummy argument list of a function or subroutine in a subprogram, that symbolic name does not identify an intrinsic function in the program unit. The data type identified with the symbolic name is specified in the same manner as for variables and arrays (4.1.2).

A name in an INTRINSIC statement must be the specific name of an intrinsic function.

15.3.2 Referencing an Intrinsic Function. An intrinsic function is referenced by using its reference as a primary in an expression. For each intrinsic function described in Table 5, execution of an intrinsic function reference causes the actions specified in Table 5, and the result depends on the values of the actual arguments. The resulting value is available to the expression that contains the function reference.

The actual arguments that constitute the argument list must agree in order, number, and type with the specification in Table 5 and may be any expression of the specified type.

A specific name of an intrinsic function that appears in an INTRINSIC statement may be used as an actual argument in an external procedure reference; however, the names of intrinsic functions for type conversion, lexical relationship, and for choosing the largest or smallest value must not be used as actual arguments. Note that such an appearance does not cause the intrinsic function to be classified as an external function (18.2.10).

15.3.3 Intrinsic Function Restrictions. Arguments for which the result is not mathematically defined or exceeds the numeric range of the processor cause the result of the function to become undefined.

Restrictions on the range of arguments and results for intrinsic functions are described in 15.10.1.

If a generic name is used to reference an intrinsic function, the type of the result (except for intrinsic functions performing type conversion, nearest integer, and absolute value with a complex argument) is the same as the type of the argument.

For those intrinsic functions that have more than one argument, all arguments must be of the same type.

If the specific name or generic name of an intrinsic function appears in the dummy argument list of a function or subroutine in a subprogram, that symbolic name does not identify an intrinsic function in the program unit. The data type identified with the symbolic name is specified in the same manner as for variables and arrays (4.1.2).

A name in an INTRINSIC statement must be the specific name or generic name of an intrinsic function.

15.3.2 <u>Referencing an Intrinsic Function</u>. An intrinsic function is referenced by using its reference as a primary in an expression. For each intrinsic function described in Table 5, execution of an intrinsic function reference causes the actions specified in Table 5, and the result depends on the values of the actual arguments. The resulting value is available to the expression that contains the function reference.

The actual arguments that constitute the argument list must agree in order, number, and type with the specification in Table 5 and may be any expression of the specified type. An actual argument in an intrinsic function reference may be any expression except a character expression involving concatenation of an operand whose length specification is an asterisk in parentheses unless the operand is the symbolic name of a constant.

A specific name of an intrinsic function that appears in an INTRINSIC statement may be used as an actual argument in an external procedure reference; however, the names of intrinsic functions for type conversion, lexical relationship, and for choosing the largest or smallest value must not be used as actual arguments. Note that such an appearance does not cause the intrinsic function to be classified as an external function (18.2.10).

15.3.3 <u>Intrinsic Function Restrictions</u>. Arguments for which the result is not mathematically defined or exceeds the numeric range of the processor cause the result of the function to become undefined.

Restrictions on the range of arguments and results for intrinsic functions are described in 15.10.1.

15.4 Statement Function

A statement function is a procedure specified by a single statement that is similar in form to an arithmetic or logical assignment statement. A statement function statement must appear only after the specification statements and before the first executable statement of the program unit in which it is referenced (3.5).

A statement function statement is classified as a nonexecutable statement; it is not a part of the normal execution sequence.

15.4.1 Form of a Statement Function Statement. The form of a statement function statement is:

fun ([*d* [,*d*]...]) = *e*

where: *fun* is the symbolic name of the statement function

d is a statement function dummy argument

e is an expression

The relationship between *fun* and *e* must conform to the assignment rules in 10.1 and 10.2. Note that the type of the expression may be different from the type of the statement function name.

Each *d* is a variable name called a statement function dummy argument. The statement function dummy argument list serves only to indicate order, number, and type of arguments for the statement function. The variable names that appear as dummy arguments of a statement function have a scope of that statement (18.1). A given symbolic name may appear only once in any statement function dummy argument list. The symbolic name of a statement function dummy argument may be used to identify other dummy arguments of the same type in different statement function statements. The name may also be used to identify a variable of the same type appearing elsewhere in the program unit, including its appearance as a dummy argument in a FUNCTION or SUBROUTINE statement. The name must not be used to identify any other entity in the program unit except a common block.

Each primary of the expression *e* must be one of the following:

(1) A constant

(2) A variable reference

(3) An array element reference

15.4 Statement Function

A statement function is a procedure specified by a single statement that is similar in form to an arithmetic, logical, or character assignment statement. A statement function statement must appear only after the specification statements and before the first executable statement of the program unit in which it is referenced (3.5).

A statement function statement is classified as a nonexecutable statement; it is not a part of the normal execution sequence.

15.4.1 Form of a Statement Function Statement. The form of a statement function statement is:

fun ([*d* [,*d*]...]) = *e*

where: *fun* is the symbolic name of the statement function

d is a statement function dummy argument

e is an expression

The relationship between *fun* and *e* must conform to the assignment rules in 10.1, 10.2, and 10.4. Note that the type of the expression may be different from the type of the statement function name.

Each *d* is a variable name called a *statement function dummy argument*. The statement function dummy argument list serves only to indicate order, number, and type of arguments for the statement function. The variable names that appear as dummy arguments of a statement function have a scope of that statement (18.1). A given symbolic name may appear only once in any statement function dummy argument list. The symbolic name of a statement function dummy argument may be used to identify other dummy arguments of the same type in different statement function statements. The name may also be used to identify a variable of the same type appearing elsewhere in the program unit, including its appearance as a dummy argument in a FUNCTION, SUBROUTINE, or ENTRY statement. The name must not be used to identify any other entity in the program unit except a common block.

Each primary of the expression *e* must be one of the following:

(1) A constant

(2) The symbolic name of a constant

(3) A variable reference

(4) An array element reference

(4) An intrinsic function reference

(5) A reference to a statement function for which the statement function statement appears in preceding lines of the program unit

(6) An external function reference

(7) A dummy procedure reference

(8) An expression enclosed in parentheses that meets all of the requirements specified for the expression _e_

Each variable reference may be either a reference to a dummy argument of the statement function or a reference to a variable that appears within the same program unit as the statement function statement.

If a statement function dummy argument name is the same as the name of another entity, the appearance of that name in the expression of a statement function statement is a reference to the statement function dummy argument. A dummy argument that appears in a FUNCTION or SUBROUTINE statement may be referenced in the expression of a statement function statement within the subprogram.

15.4.2 Referencing a Statement Function. A statement function is referenced by using its function reference as a primary in an expression.

Execution of a statement function reference results in:

(1) evaluation of actual arguments that are expressions,

(2) association of actual arguments with the corresponding dummy arguments,

(3) evaluation of the expression _e_, and

(4) conversion, if necessary, of an arithmetic expression value to the type of the statement function according to the assignment rules in 10.1.

The resulting value is available to the expression that contains the function reference.

The actual arguments, which constitute the argument list, must agree in order, number, and type with the corresponding dummy arguments. An actual argument in a statement function reference may be any expression.

(5) An intrinsic function reference

(6) A reference to a statement function for which the statement function statement appears in preceding lines of the program unit

(7) An external function reference

(8) A dummy procedure reference

(9) An expression enclosed in parentheses that meets all of the requirements specified for the expression _e_

Each variable reference may be either a reference to a dummy argument of the statement function or a reference to a variable that appears within the same program unit as the statement function statement.

If a statement function dummy argument name is the same as the name of another entity, the appearance of that name in the expression of a statement function statement is a reference to the statement function dummy argument. A dummy argument that appears in a FUNCTION or SUBROUTINE statement may be referenced in the expression of a statement function statement within the subprogram. A dummy argument that appears in an ENTRY statement that precedes a statement function statement may be referenced in the expression of the statement function statement within the subprogram.

15.4.2 Referencing a Statement Function. A statement function is referenced by using its function reference as a primary in an expression.

Execution of a statement function reference results in:

(1) evaluation of actual arguments that are expressions,

(2) association of actual arguments with the corresponding dummy arguments,

(3) evaluation of the expression _e_, and

(4) conversion, if necessary, of an arithmetic expression value to the type of the statement function according to the assignment rules in 10.1 or a change, if necessary, in the length of a character expression value according to the rules in 10.4.

The resulting value is available to the expression that contains the function reference.

The actual arguments, which constitute the argument list, must agree in order, number, and type with the corresponding dummy arguments. An actual argument in a statement function reference may be any expression except a character expression involving concatenation of an operand whose

When a statement function reference is executed, its actual arguments must be defined.

15.4.3 Statement Function Restrictions. A statement function may be referenced only in the program unit that contains the statement function statement.

A statement function statement must not contain a reference to another statement function that appears following the reference in the sequence of lines in the program unit. The symbolic name used to identify a statement function must not appear as a symbolic name in any specification statement except in a type-statement (to specify the type of the function) or as the name of a common block in the same program unit.

An external function reference in the expression of a statement function statement must not cause a dummy argument of the statement function to become undefined or redefined.

The symbolic name of a statement function is a local name (18.1.2) and must not be the same as the name of any other entity in the program unit except the name of a common block.

The symbolic name of a statement function may not be an actual argument. It must not appear in an EXTERNAL statement.

A statement function statement in a function subprogram must not contain a function reference to the name of the function subprogram.

A statement function must not be of type character.

The length specification of a statement function dummy argument of type character must be an integer constant.

15.5 External Functions

An external function is specified externally to the program unit that references it. An external function is a procedure and may be specified in a function subprogram or by some other means.

15.5.1 Function Subprogram and FUNCTION Statement. A function subprogram specifies an external function. A function subprogram is a program unit that has a FUNCTION statement as its first statement. The form of a function subprogram is as described in 2.4 and 3.5, except as noted in 15.5.3 and 15.7.4.

length specification is an asterisk in parentheses unless the operand is the symbolic name of a constant.

When a statement function reference is executed, its actual arguments must be defined.

15.4.3 <u>Statement Function Restrictions</u>. A statement function may be referenced only in the program unit that contains the statement function statement.

A statement function statement must not contain a reference to another statement function that appears following the reference in the sequence of lines in the program unit. The symbolic name used to identify a statement function must not appear as a symbolic name in any specification statement except in a type-statement (to specify the type of the function) or as the name of a common block in the same program unit.

An external function reference in the expression of a statement function statement must not cause a dummy argument of the statement function to become undefined or redefined.

The symbolic name of a statement function is a local name (18.1.2) and must not be the same as the name of any other entity in the program unit except the name of a common block.

The symbolic name of a statement function may not be an actual argument. It must not appear in an EXTERNAL statement.

A statement function statement in a function subprogram must not contain a function reference to the name of the function subprogram or an entry name in the function subprogram.

The length specification of a character statement function or statement function dummy argument of type character must be an integer constant expression.

15.5 <u>External Functions</u>

An external function is specified externally to the program unit that references it. An external function is a procedure and may be specified in a function subprogram or by some other means.

15.5.1 <u>Function Subprogram and FUNCTION Statement</u>. A function subprogram specifies one or more external functions (15.7). A function subprogram is a program unit that has a FUNCTION statement as its first statement. The form of a function subprogram is as described in 2.4 and 3.5, except as noted in 15.5.3 and 15.7.4.

The form of a FUNCTION statement is:

[typ] FUNCTION fun ([d [,d]...])

where: typ is one of INTEGER, REAL, or LOGICAL

fun is the symbolic name of the function subprogram in which the FUNCTION statement appears. fun is an external function name.

d is a variable name, array name, or dummy procedure name. d is a dummy argument.

The symbolic name of a function subprogram must appear as a variable name in the function subprogram. During every execution of the external function, this variable must become defined and, once defined, may be referenced or become redefined. The value of this variable when a RETURN or END statement is executed in the subprogram is the value of the function.

An external function in a function subprogram may define one or more of its dummy arguments to return values in addition to the value of the function.

15.5.2 Referencing an External Function. An external function is referenced by using its reference as a primary in an expression.

15.5.2.1 Execution of an External Function Reference. Execution of an external function reference results in:

(1) evaluation of actual arguments that are expressions,

(2) association of actual arguments with the corresponding dummy arguments, and

(3) the actions specified by the referenced function.

The type of the function name in the function reference must be the same as the type of the function name in the referenced function. Note that an external function must not be of type character.

The form of a FUNCTION statement is:

[typ] FUNCTION fun ([d [,d]...])

where: typ is one of INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, or CHARACTER [*len] where len is the length specification of the result of the character function. len may have any of the forms allowed in a CHARACTER statement (8.4.2) except that an integer constant expression must not include the symbolic name of a constant. If a length is not specified in a CHARACTER FUNCTION statement, the character function has a length of one.

fun is the symbolic name of the function subprogram in which the FUNCTION statement appears. fun is an external function name.

d is a variable name, array name, or dummy procedure name. d is a dummy argument.

The symbolic name of a function subprogram or an associated entry name of the same type must appear as a variable name in the function subprogram. During every execution of the external function, this variable must become defined and, once defined, may be referenced or become redefined. The value of this variable when a RETURN or END statement is executed in the subprogram is the value of the function. If this variable is a character variable with a length specification that is an asterisk in parentheses, it must not appear as an operand for concatenation except in a character assignment statement (10.4).

An external function in a function subprogram may define one or more of its dummy arguments to return values in addition to the value of the function.

15.5.2 Referencing an External Function. An external function is referenced by using its reference as a primary in an expression.

15.5.2.1 Execution of an External Function Reference. Execution of an external function reference results in:

(1) evaluation of actual arguments that are expressions,

(2) association of actual arguments with the corresponding dummy arguments, and

(3) the actions specified by the referenced function.

The type of the function name in the function reference must be the same as the type of the function name in the referenced function. The length of the character function

When an external function reference is executed, the function must be one of the external functions in the executable program.

15.5.2.2 Actual Arguments for an External Function. The actual arguments in an external function reference must agree in order, number, and type with the corresponding dummy arguments in the referenced function. The use of a subroutine name as an actual argument is an exception to the rule requiring agreement of type because subroutine names do not have a type.

An actual argument in an external function reference must be one of the following:

(1) An expression

(2) An array name

(3) An intrinsic function name

(4) An external procedure name

(5) A dummy procedure name

Note that an actual argument in a function reference may be a dummy argument that appears in a dummy argument list within the subprogram containing the reference.

15.5.3 Function Subprogram Restrictions. A FUNCTION statement must appear only as the first statement of a function subprogram. A function subprogram may contain any other statement except a SUBROUTINE or PROGRAM statement.

The symbolic name of an external function is a global name (18.1.1) and must not be the same as any other global name or any local name, except a variable name, in the function subprogram.

Within a function subprogram, the symbolic name of a function specified by the FUNCTION statement must not appear in any other nonexecutable statement, except a type-statement. In an executable statement, such a name may appear only as a variable.

If the type of a function is specified in a FUNCTION statement, the function name must not appear in a type-statement. Note that a name must not have its type explicitly specified more than once in a program unit.

in a character function reference must be the same as the length of the character function in the referenced function.

When an external function reference is executed, the function must be one of the external functions in the executable program.

15.5.2.2 <u>Actual Arguments for an External Function</u>. The actual arguments in an external function reference must agree in order, number, and type with the corresponding dummy arguments in the referenced function. The use of a subroutine name as an actual argument is an exception to the rule requiring agreement of type because subroutine names do not have a type.

An actual argument in an external function reference must be one of the following:

(1) An expression except a character expression involving concatenation of an operand whose length specification is an asterisk in parentheses unless the operand is the symbolic name of a constant

(2) An array name

(3) An intrinsic function name

(4) An external procedure name

(5) A dummy procedure name

Note that an actual argument in a function reference may be a dummy argument that appears in a dummy argument list within the subprogram containing the reference.

15.5.3 <u>Function Subprogram Restrictions</u>. A FUNCTION statement must appear only as the first statement of a function subprogram. A function subprogram may contain any other statement except a BLOCK DATA, SUBROUTINE, or PROGRAM statement.

The symbolic name of an external function is a global name (18.1.1) and must not be the same as any other global name or any local name, except a variable name, in the function subprogram.

Within a function subprogram, the symbolic name of a function specified by a FUNCTION or ENTRY statement must not appear in any other nonexecutable statement, except a type-statement. In an executable statement, such a name may appear only as a variable.

If the type of a function is specified in a FUNCTION statement, the function name must not appear in a type-statement. Note that a name must not have its type explicitly specified more than once in a program unit.

A function subprogram name must not be of type character.

In a function subprogram, the symbolic name of a dummy argument is local to the program unit and must not appear in an EQUIVALENCE, SAVE, INTRINSIC, DATA, or COMMON statement, except as a common block name.

A function specified in a subprogram may be referenced within any other procedure subprogram or the main program of the executable program. A function subprogram must not reference itself, either directly or indirectly.

15.6 Subroutines

A subroutine is specified externally to the program unit that references it. A subroutine is a procedure and may be specified in a subroutine subprogram or by some other means.

15.6.1 Subroutine Subprogram and SUBROUTINE Statement. A subroutine subprogram specifies a subroutine. A subroutine subprogram is a program unit that has a SUBROUTINE statement as its first statement. The form of a subroutine subprogram is as described in 2.4 and 3.5, except as noted in 15.6.3 and 15.7.4.

The form of a SUBROUTINE statement is:

SUBROUTINE sub [([d [,d]...])]

where: sub is the symbolic name of the subroutine subprogram in which the SUBROUTINE statement appears. sub is a subroutine name.

d is a variable name, array name, or dummy procedure name. d is a dummy argument.

Note that if there are no dummy arguments, either of the forms sub or sub() may be used in the SUBROUTINE statement. A subroutine that is specified by either form may be referenced by a CALL statement of the form CALL sub or CALL sub().

One or more dummy arguments of a subroutine in a subprogram may become defined or redefined to return results.

If the name of a function subprogram is of type character, each entry name in the function subprogram must be of type character. If the name of the function subprogram or any entry in the subprogram has a length of (*) declared, all such entities must have a length of (*) declared; otherwise, all such entities must have a length specification of the same integer value.

In a function subprogram, the symbolic name of a dummy argument is local to the program unit and must not appear in an EQUIVALENCE, PARAMETER, SAVE, INTRINSIC, DATA, or COMMON statement, except as a common block name.

A character dummy argument whose length specification is an asterisk in parentheses must not appear as an operand for concatenation, except in a character assignment statement (10.4).

A function specified in a subprogram may be referenced within any other procedure subprogram or the main program of the executable program. A function subprogram must not reference itself, either directly or indirectly.

15.6 <u>Subroutines</u>

A subroutine is specified externally to the program unit that references it. A subroutine is a procedure and may be specified in a subroutine subprogram or by some other means.

15.6.1 <u>Subroutine Subprogram and SUBROUTINE Statement</u>. A subroutine subprogram specifies one or more subroutines (15.7). A subroutine subprogram is a program unit that has a SUBROUTINE statement as its first statement. The form of a subroutine subprogram is as described in 2.4 and 3.5, except as noted in 15.6.3 and 15.7.4.

The form of a SUBROUTINE statement is:

SUBROUTINE <u>sub</u> [([<u>d</u> [,<u>d</u>]...])]

where: <u>sub</u> is the symbolic name of the subroutine subprogram in which the SUBROUTINE statement appears. <u>sub</u> is a <u>subroutine</u> <u>name</u>.

<u>d</u> is a variable name, array name, or dummy procedure name, or is an asterisk (15.9.3.5). <u>d</u> is a dummy argument.

Note that if there are no dummy arguments, either of the forms <u>sub</u> or <u>sub</u>() may be used in the SUBROUTINE statement. A subroutine that is specified by either form may be referenced by a CALL statement of the form CALL <u>sub</u> or CALL <u>sub</u>().

One or more dummy arguments of a subroutine in a subprogram may become defined or redefined to return results.

15.6.2 Subroutine Reference. A subroutine is referenced by a CALL statement.

15.6.2.1 Form of a CALL Statement. The form of a CALL statement is:

CALL sub [([a [,a]...])]

where: sub is the symbolic name of a subroutine or dummy procedure

a is an actual argument

15.6.2.2 Execution of a CALL Statement. Execution of a CALL statement results in

(1) evaluation of actual arguments that are expressions,

(2) association of actual arguments with the corresponding dummy arguments, and

(3) the actions specified by the referenced subroutine.

Return of control from the referenced subroutine completes execution of the CALL statement.

A subroutine specified in a subprogram may be referenced within any other procedure subprogram or the main program of the executable program. A subprogram must not reference itself, either directly or indirectly.

When a CALL statement is executed, the referenced subroutine must be one of the subroutines specified in subroutine subprograms or by other means in the executable program.

15.6.2.3 Actual Arguments for a Subroutine. The actual arguments in a subroutine reference must agree in order, number, and type with the corresponding dummy arguments in the dummy argument list of the referenced subroutine. The use of a subroutine name as an actual argument is an exception to the rule requiring agreement of type.

An actual argument in a subroutine reference must be one of the following:

(1) An expression

(2) An array name

(3) An intrinsic function name

(4) An external procedure name

15.6.2 <u>Subroutine Reference</u>. A subroutine is referenced by a CALL statement.

15.6.2.1 <u>Form of a CALL Statement</u>. The form of a CALL statement is:

CALL <u>sub</u> [([<u>a</u> [,<u>a</u>]...])]

where: <u>sub</u> is the symbolic name of a subroutine or dummy procedure

<u>a</u> is an actual argument

15.6.2.2 <u>Execution of a CALL Statement</u>. Execution of a CALL statement results in

(1) evaluation of actual arguments that are expressions,

(2) association of actual arguments with the corresponding dummy arguments, and

(3) the actions specified by the referenced subroutine.

Return of control from the referenced subroutine completes execution of the CALL statement.

A subroutine specified in a subprogram may be referenced within any other procedure subprogram or the main program of the executable program. A subprogram must not reference itself, either directly or indirectly.

When a CALL statement is executed, the referenced subroutine must be one of the subroutines specified in subroutine subprograms or by other means in the executable program.

15.6.2.3 <u>Actual Arguments for a Subroutine</u>. The actual arguments in a subroutine reference must agree in order, number, and type with the corresponding dummy arguments in the dummy argument list of the referenced subroutine. The use of a subroutine name or an alternate return specifier as an actual argument is an exception to the rule requiring agreement of type.

An actual argument in a subroutine reference must be one of the following:

(1) An expression except a character expression involving concatenation of an operand whose length specification is an asterisk in parentheses unless the operand is the symbolic name of a constant

(2) An array name

(3) An intrinsic function name

(4) An external procedure name

(5) A dummy procedure name

Note that an actual argument in a subroutine reference may be a dummy argument name that appears in a dummy argument list within the subprogram containing the reference.

15.6.3 Subroutine Subprogram Restrictions. A SUBROUTINE statement must appear only as the first statement of a subroutine subprogram. A subroutine subprogram may contain any other statement except a FUNCTION or PROGRAM statement.

The symbolic name of a subroutine is a global name (18.1.1) and must not be the same as any other global name or any local name in the program unit.

In a subroutine subprogram, the symbolic name of a dummy argument is local to the program unit and must not appear in an EQUIVALENCE, SAVE, INTRINSIC, DATA, or COMMON statement, except as a common block name.

15.7 ENTRY Statement

The ENTRY statement is not included in the subset.

15.7.1 Form of an ENTRY Statement. The ENTRY statement is not included in the subset.

(5) A dummy procedure name

(6) An alternate return specifier, of the form *s, where s is the statement label of an executable statement that appears in the same program unit as the CALL statement (15.8.3)

Note that an actual argument in a subroutine reference may be a dummy argument name that appears in a dummy argument list within the subprogram containing the reference. An asterisk dummy argument must not be used as an actual argument in a subprogram reference.

15.6.3 Subroutine Subprogram Restrictions. A SUBROUTINE statement must appear only as the first statement of a subroutine subprogram. A subroutine subprogram may contain any other statement except a BLOCK DATA, FUNCTION, or PROGRAM statement.

The symbolic name of a subroutine is a global name (18.1.1) and must not be the same as any other global name or any local name in the program unit.

In a subroutine subprogram, the symbolic name of a dummy argument is local to the program unit and must not appear in an EQUIVALENCE, PARAMETER, SAVE, INTRINSIC, DATA, or COMMON statement, except as a common block name.

A character dummy argument whose length specification is an asterisk in parentheses must not appear as an operand for concatenation, except in a character assignment statement (10.4).

15.7 ENTRY Statement

An ENTRY statement permits a procedure reference to begin with a particular executable statement within the function or subroutine subprogram in which the ENTRY statement appears. It may appear anywhere within a function subprogram after the FUNCTION statement or within a subroutine subprogram after the SUBROUTINE statement, except that an ENTRY statement must not appear between a block IF statement and its corresponding END IF statement, or between a DO statement and the terminal statement of its DO-loop.

Optionally, a subprogram may have one or more ENTRY statements.

An ENTRY statement is classified as a nonexecutable statement.

15.7.1 Form of an ENTRY Statement. The form of an ENTRY statement is:

```
ENTRY en [( [d [,d]...] )]
```

15.7.2 Referencing External Procedure by Entry Name. The ENTRY statement is not included in the subset.

15.7.3 Entry Association. The ENTRY statement is not included in the subset.

where: *en* is the symbolic name of an entry in a function or subroutine subprogram and is called an *entry name*. If the ENTRY statement appears in a subroutine subprogram, *en* is a *subroutine name*. If the ENTRY statement appears in a function subprogram, *en* is an *external function name*.

d is a variable name, array name, or dummy procedure name, or is an asterisk. *d* is a dummy argument. An asterisk is permitted in an ENTRY statement only in a subroutine subprogram.

Note that if there are no dummy arguments, either of the forms *en* or *en*() may be used in the ENTRY statement. A function that is specified by either form must be referenced by the form *en*() (15.2.1). A subroutine that is specified by either form may be referenced by a CALL statement of the form CALL *en* or CALL *en*().

The entry name *en* in a function subprogram may appear in a type-statement.

15.7.2 *Referencing External Procedure by Entry Name*. An entry name in an ENTRY statement in a function subprogram identifies an external function within the executable program and may be referenced as an external function (15.5.2). An entry name in an ENTRY statement in a subroutine subprogram identifies a subroutine within the executable program and may be referenced as a subroutine (15.6.2).

When an entry name *en* is used to reference a procedure, execution of the procedure begins with the first executable statement that follows the ENTRY statement whose entry name is *en*.

An entry name is available for reference in any program unit of an executable program, except in the program unit that contains the entry name in an ENTRY statement.

The order, number, type, and names of the dummy arguments in an ENTRY statement may be different from the order, number, type, and names of the dummy arguments in the FUNCTION statement or SUBROUTINE statement and other ENTRY statements in the same subprogram. However, each reference to a function or subroutine must use an actual argument list that agrees in order, number, and type with the dummy argument list in the corresponding FUNCTION, SUBROUTINE, or ENTRY statement. The use of a subroutine name or an alternate return specifier as an actual argument is an exception to the rule requiring agreement of type.

15.7.3 *Entry Association*. Within a function subprogram, all variables whose names are also the names of entries are associated with each other and with the variable, if any, whose name is also the name of the function subprogram

15.7.4 ENTRY Statement Restrictions. The ENTRY statement is not included in the subset.

15.8 RETURN Statement

A RETURN statement causes return of control to the referencing program unit and may appear only in a function subprogram or subroutine subprogram.

(17.1.3). Therefore, any such variable that becomes defined causes all associated variables of the same type to become defined and all associated variables of different type to become undefined. Such variables are not required to be of the same type unless the type is character, but the variable whose name is used to reference the function must be in a defined state when a RETURN or END statement is executed in the subprogram. An associated variable of a different type must not become defined during the execution of the function reference.

15.7.4 <u>ENTRY Statement Restrictions</u>. Within a subprogram, an entry name must not appear both as an entry name in an ENTRY statement and as a dummy argument in a FUNCTION, SUBROUTINE, or ENTRY statement and must not appear in an EXTERNAL statement.

In a function subprogram, a variable name that is the same as an entry name must not appear in any statement that precedes the appearance of the entry name in an ENTRY statement, except in a type-statement.

If an entry name in a function subprogram is of type character, each entry name and the name of the function subprogram must be of type character. If the name of the function subprogram or any entry in the subprogram has a length of (*) declared, all such entities must have a length of (*) declared; otherwise, all such entities must have a length specification of the same integer value.

In a subprogram, a name that appears as a dummy argument in an ENTRY statement must not appear in an executable statement preceding that ENTRY statement unless it also appears in a FUNCTION, SUBROUTINE, or ENTRY statement that precedes the executable statement.

In a subprogram, a name that appears as a dummy argument in an ENTRY statement must not appear in the expression of a statement function statement unless the name is also a dummy argument of the statement function, appears in a FUNCTION or SUBROUTINE statement, or appears in an ENTRY statement that precedes the statement function statement.

If a dummy argument appears in an executable statement, the execution of the executable statement is permitted during the execution of a reference to the function or subroutine only if the dummy argument appears in the dummy argument list of the procedure name referenced. Note that the association of dummy arguments with actual arguments is not retained between references to a function or subroutine.

15.8 <u>RETURN Statement</u>

A RETURN statement causes return of control to the referencing program unit and may appear only in a function subprogram or subroutine subprogram.

15.8.1 Form of a RETURN Statement. The form of a RETURN statement is:

RETURN

15.8.2 Execution of a RETURN Statement. Execution of a RETURN statement terminates the reference of a function or subroutine subprogram. Such subprograms may contain more than one RETURN statement; however, a subprogram need not contain a RETURN statement. Execution of an END statement in a function or subroutine subprogram has the same effect as executing a RETURN statement in the subprogram.

In the execution of an executable program, a function or subroutine subprogram must not be referenced a second time without the prior execution of a RETURN or END statement in that procedure.

Execution of a RETURN statement in a function subprogram causes return of control to the currently referencing program unit. The value of the function (15.5) must be defined and is available to the referencing program unit.

Execution of a RETURN statement in a subroutine subprogram causes return of control to the currently referencing program unit. Return of control to the referencing program unit completes execution of the CALL statement.

Execution of a RETURN statement terminates the association between the dummy arguments of the external procedure in the subprogram and the current actual arguments.

15.8.3 Alternate Return. Alternate return is not included in the subset.

15.8.1 Form of a RETURN Statement. The form of a RETURN statement in a function subprogram is:

RETURN

The form of a RETURN statement in a subroutine subprogram is:

RETURN [*e*]

where *e* is an integer expression.

15.8.2 Execution of a RETURN Statement. Execution of a RETURN statement terminates the reference of a function or subroutine subprogram. Such subprograms may contain more than one RETURN statement; however, a subprogram need not contain a RETURN statement. Execution of an END statement in a function or subroutine subprogram has the same effect as executing a RETURN statement in the subprogram.

In the execution of an executable program, a function or subroutine subprogram must not be referenced a second time without the prior execution of a RETURN or END statement in that procedure.

Execution of a RETURN statement in a function subprogram causes return of control to the currently referencing program unit. The value of the function (15.5) must be defined and is available to the referencing program unit.

Execution of a RETURN statement in a subroutine subprogram causes return of control to the currently referencing program unit. Return of control to the referencing program unit completes execution of the CALL statement.

Execution of a RETURN statement terminates the association between the dummy arguments of the external procedure in the subprogram and the current actual arguments.

15.8.3 Alternate Return. If *e* is not specified in a RETURN statement, or if the value of *e* is less than one or greater than the number of asterisks in the SUBROUTINE or subroutine ENTRY statement that specifies the currently referenced name, control returns to the CALL statement that initiated the subprogram reference and this completes the execution of the CALL statement.

If $1 \le e \le n$, where *n* is the number of asterisks in the SUBROUTINE or subroutine ENTRY statement that specifies the currently referenced name, the value of *e* identifies the *e*th asterisk in the dummy argument list. Control is returned to the statement identified by the alternate return specifier in the CALL statement that is associated with the *e*th asterisk in the dummy argument list of the currently referenced name. This completes the execution of the CALL statement.

15.8.4 Definition Status. Execution of a RETURN statement (or END statement) within a subprogram causes all entities within the subprogram to become undefined, except for the following:

(1) Entities specified by SAVE statements

(2) Entities in blank common

(3) Initially defined entities that have neither been redefined or become undefined

(4) Entities in a named common block that appears in the subprogram and appears in at least one other program unit that is referencing, either directly or indirectly, the subprogram

Note that if a named common block appears in the main program, the entities in the named common block do not become undefined at the execution of any RETURN statement in the executable program.

15.9 Arguments and Common Blocks

Arguments and common blocks provide means of communication between the referencing program unit and the referenced procedure.

Data may be communicated to a statement function or intrinsic function by an argument list. Data may be communicated to and from an external procedure by an argument list or common blocks. Procedure names may be communicated to an external procedure only by an argument list.

A dummy argument appears in the argument list of a procedure. An actual argument appears in the argument list of a procedure reference.

The number of actual arguments must be the same as the number of dummy arguments in the procedure referenced.

15.9.1 Dummy Arguments. Statement functions, function subprograms, and subroutine subprograms use dummy arguments to indicate the types of actual arguments and whether each argument is a single value, array of values, or procedure. Note that a statement function dummy argument may be only a variable.

Each dummy argument is classified as a variable, array, or dummy procedure. Dummy argument names may appear wherever an actual name of the same class (Section 18) and type may appear, except where they are explicitly prohibited.

15.8.4 <u>Definition Status</u>. Execution of a RETURN statement (or END statement) within a subprogram causes all entities within the subprogram to become undefined, except for the following:

(1) Entities specified by SAVE statements

(2) Entities in blank common

(3) Initially defined entities that have neither been redefined or become undefined

(4) Entities in a named common block that appears in the subprogram and appears in at least one other program unit that is referencing, either directly or indirectly, the subprogram

Note that if a named common block appears in the main program, the entities in the named common block do not become undefined at the execution of any RETURN statement in the executable program.

15.9 <u>Arguments and Common Blocks</u>

Arguments and common blocks provide means of communication between the referencing program unit and the referenced procedure.

Data may be communicated to a statement function or intrinsic function by an argument list. Data may be communicated to and from an external procedure by an argument list or common blocks. Procedure names may be communicated to an external procedure only by an argument list.

A dummy argument appears in the argument list of a procedure. An actual argument appears in the argument list of a procedure reference.

The number of actual arguments must be the same as the number of dummy arguments in the procedure referenced.

15.9.1 <u>Dummy Arguments</u>. Statement functions, function subprograms, and subroutine subprograms use dummy arguments to indicate the types of actual arguments and whether each argument is a single value, array of values, procedure, or statement label. Note that a statement function dummy argument may be only a variable.

Each dummy argument is classified as a variable, array, dummy procedure, or asterisk. Dummy argument names may appear wherever an actual name of the same class (Section 18) and type may appear, except where they are explicitly prohibited.

Dummy argument names of type integer may appear in adjustable dimensions in dummy array declarators (5.5.1). Dummy argument names must not appear in EQUIVALENCE, DATA, SAVE, INTRINSIC, or COMMON statements, except as common block names. A dummy argument name must not be the same as the procedure name appearing in a FUNCTION, SUBROUTINE, or statement function statement in the same program unit.

15.9.2 <u>Actual Arguments</u>. Actual arguments specify the entities that are to be associated with the dummy arguments for a particular reference of a subroutine or function. An actual argument must not be the name of a statement function in the program unit containing the reference. Actual arguments may be constants, function references, expressions involving operators, and expressions enclosed in parentheses if and only if the associated dummy argument is a variable that is not defined during execution of the referenced external procedure.

The type of each actual argument must agree with the type of its associated dummy argument, except when the actual argument is a subroutine name (15.9.3.4).

15.9.3 <u>Association of Dummy and Actual Arguments</u>. At the execution of a function or subroutine reference, an association is established between the corresponding dummy and actual arguments. The first dummy argument becomes associated with the first actual argument, the second dummy argument becomes associated with the second actual argument, etc.

All appearances within a function or subroutine subprogram of a dummy argument whose name appears in the dummy argument list of the procedure name referenced become associated with the actual argument when a reference to the function or subroutine is executed.

A valid association occurs only if the type of the actual argument is the same as the type of the corresponding dummy argument. A subroutine name has no type and must be associated with a dummy procedure name.

If an actual argument is an expression, it is evaluated just before the association of arguments takes place.

If an actual argument is an array element name, its subscript is evaluated just before the association of arguments takes place. Note that the subscript value remains constant as long as that association of arguments persists, even if the subscript contains variables that are redefined during the association.

Dummy argument names of type integer may appear in adjustable dimensions in dummy array declarators (5.5.1). Dummy argument names must not appear in EQUIVALENCE, DATA, PARAMETER, SAVE, INTRINSIC, or COMMON statements, except as common block names. A dummy argument name must not be the same as the procedure name appearing in a FUNCTION, SUBROUTINE, ENTRY, or statement function statement in the same program unit.

15.9.2 Actual Arguments. Actual arguments specify the entities that are to be associated with the dummy arguments for a particular reference of a subroutine or function. An actual argument must not be the name of a statement function in the program unit containing the reference. Actual arguments may be constants, symbolic names of constants, function references, expressions involving operators, and expressions enclosed in parentheses if and only if the associated dummy argument is a variable that is not defined during execution of the referenced external procedure.

The type of each actual argument must agree with the type of its associated dummy argument, except when the actual argument is a subroutine name (15.9.3.4) or an alternate return specifier (15.6.2.3).

15.9.3 Association of Dummy and Actual Arguments. At the execution of a function or subroutine reference, an association is established between the corresponding dummy and actual arguments. The first dummy argument becomes associated with the first actual argument, the second dummy argument becomes associated with the second actual argument, etc.

All appearances within a function or subroutine subprogram of a dummy argument whose name appears in the dummy argument list of the procedure name referenced become associated with the actual argument when a reference to the function or subroutine is executed.

A valid association occurs only if the type of the actual argument is the same as the type of the corresponding dummy argument. A subroutine name has no type and must be associated with a dummy procedure name. An alternate return specifier has no type and must be associated with an asterisk.

If an actual argument is an expression, it is evaluated just before the association of arguments takes place.

If an actual argument is an array element name, its subscript is evaluated just before the association of arguments takes place. Note that the subscript value remains constant as long as that association of arguments persists, even if the subscript contains variables that are redefined during the association.

If an actual argument is an external procedure name, the procedure must be available at the time a reference to it is executed.

If an actual argument becomes associated with a dummy argument that appears in an adjustable dimension (5.5.1), the actual argument must be defined with an integer value at the time the procedure is referenced.

A dummy argument is undefined if it is not currently associated with an actual argument. An adjustable array is undefined if the dummy argument array is not currently associated with an actual argument array or if any variable appearing in the adjustable array declarator is not currently associated with an actual argument and is not in a common block.

Argument association may be carried through more than one level of procedure reference. A valid association exists at the last level only if a valid association exists at all intermediate levels. Argument association within a program unit terminates at the execution of a RETURN or END statement in the program unit. Note that there is no retention of argument association between one reference of a subprogram and the next reference of the subprogram.

15.9.3.1 <u>Length of Character Dummy and Actual Arguments</u>. If a dummy argument is of type character, the associated actual argument must be of type character and the length of the dummy argument must be equal to the length of the actual argument.

If a dummy argument of type character is an array name, the restriction on length is for each array element.

If an actual argument is a character substring name, its substring expressions are evaluated just before the association of arguments takes place. Note that the value of each of the substring expressions remains constant as long as that association of arguments persists, even if the substring expression contains variables that are redefined during the association.

If an actual argument is an external procedure name, the procedure must be available at the time a reference to it is executed.

If an actual argument becomes associated with a dummy argument that appears in an adjustable dimension (5.5.1), the actual argument must be defined with an integer value at the time the procedure is referenced.

A dummy argument is undefined if it is not currently associated with an actual argument. An adjustable array is undefined if the dummy argument array is not currently associated with an actual argument array or if any variable appearing in the adjustable array declarator is not currently associated with an actual argument and is not in a common block.

Argument association may be carried through more than one level of procedure reference. A valid association exists at the last level only if a valid association exists at all intermediate levels. Argument association within a program unit terminates at the execution of a RETURN or END statement in the program unit. Note that there is no retention of argument association between one reference of a subprogram and the next reference of the subprogram.

15.9.3.1 <u>Length of Character Dummy and Actual Arguments</u>. If a dummy argument is of type character, the associated actual argument must be of type character and the length of the dummy argument must be less than or equal to the length of the actual argument. If the length <u>len</u> of a dummy argument of type character is less than the length of an associated actual argument, the leftmost <u>len</u> characters of the actual argument are associated with the dummy argument.

If a dummy argument of type character is an array name, the restriction on length is for the entire array and not for each array element. The length of an array element in the dummy argument array may be different from the length of an array element in an associated actual argument array, array element, or array element substring, but the dummy argument array must not extend beyond the end of the associated actual argument array.

If an actual argument is a character substring, the length of the actual argument is the length of the substring. If an actual argument is the concatenation of two or more

15.9.3.2 Variables as Dummy Arguments. A dummy argument that is a variable may be associated with an actual argument that is a variable, array element, or expression.

If the actual argument is a variable name or array element name, the associated dummy argument may be defined or redefined within the subprogram. If the actual argument is a constant, a function reference, an expression involving operators, or an expression enclosed in parentheses, the associated dummy argument must not be redefined within the subprogram.

15.9.3.3 Arrays as Dummy Arguments. Within a program unit, the array declarator given for an array provides all array declarator information needed for the array in an execution of the program unit. The number and size of dimensions in an actual argument array declarator may be different from the number and size of the dimensions in an associated dummy argument array declarator.

A dummy argument that is an array may be associated with an actual argument that is either an array or an array element.

If the actual argument is a noncharacter or character array name, the size of the dummy argument array must not exceed the size of the actual argument array, and each actual argument array element becomes associated with the dummy argument array element that has the same subscript value as the actual argument array element. Note that association by array elements exists for character arrays because there must be agreement in length between the actual argument and the dummy argument array elements.

If the actual argument is a noncharacter or character array element name, the size of the dummy argument array must not exceed the size of the actual argument array plus one minus the subscript value of the array element. When an actual argument is a noncharacter or character array element name with a subscript value of $\underline{as}$, the dummy argument array element with a subscript value of $\underline{ds}$ becomes associated with the actual argument array element that has a subscript value of $\underline{as} + \underline{ds} - 1$ (Table 1, 5.4.3).

operands, its length is the sum of the lengths of the operands.

15.9.3.2 Variables as Dummy Arguments. A dummy argument that is a variable may be associated with an actual argument that is a variable, array element, substring, or expression.

If the actual argument is a variable name, array element name, or substring name, the associated dummy argument may be defined or redefined within the subprogram. If the actual argument is a constant, a symbolic name of a constant, a function reference, an expression involving operators, or an expression enclosed in parentheses, the associated dummy argument must not be redefined within the subprogram.

15.9.3.3 Arrays as Dummy Arguments. Within a program unit, the array declarator given for an array provides all array declarator information needed for the array in an execution of the program unit. The number and size of dimensions in an actual argument array declarator may be different from the number and size of the dimensions in an associated dummy argument array declarator.

A dummy argument that is an array may be associated with an actual argument that is an array, array element, or array element substring.

If the actual argument is a noncharacter array name, the size of the dummy argument array must not exceed the size of the actual argument array, and each actual argument array element becomes associated with the dummy argument array element that has the same subscript value as the actual argument array element. Note that association by array elements exists for character arrays if there is agreement in length between the actual argument and the dummy argument array elements; if the lengths do not agree, the dummy and actual argument array elements do not consist of the same characters, but an association still exists.

If the actual argument is a noncharacter array element name, the size of the dummy argument array must not exceed the size of the actual argument array plus one minus the subscript value of the array element. When an actual argument is a noncharacter array element name with a subscript value of as, the dummy argument array element with a subscript value of ds becomes associated with the actual argument array element that has a subscript value of $as + ds - 1$ (Table 1, 5.4.3).

If the actual argument is a character array name, character array element name, or character array element substring name and begins at character storage unit acu of an array, character storage unit dcu of an associated dummy argument array becomes associated with character storage unit $acu + dcu - 1$ of the actual argument array.

15.9.3.4 Procedures as Dummy Arguments. A dummy argument that is a dummy procedure may be associated only with an actual argument that is an intrinsic function, external function, subroutine, or another dummy procedure.

If a dummy argument is used as if it were an external function, the associated actual argument must be an intrinsic function, external function, or dummy procedure. A dummy argument that becomes associated with an intrinsic function never has any automatic typing property, even if the dummy argument name appears in Table 5 (15.10). Therefore, the type of the dummy argument must agree with the type of the result of all specific actual arguments that become associated with the dummy argument. If a dummy argument name is used as if it were an external function and that name also appears in Table 5, the intrinsic function corresponding to the dummy argument name is not available for referencing within the subprogram.

A dummy argument that is used as a procedure name in a function reference and is associated with an intrinsic function must have arguments that agree in order, number, and type with those specified in Table 5 for the intrinsic function.

If a dummy argument appears in a type-statement and an EXTERNAL statement, the actual argument must be the name of an intrinsic function, external function, or dummy procedure.

If the dummy argument is referenced as a subroutine, the actual argument must be the name of a subroutine or dummy procedure and must not appear in a type-statement or be referenced as a function.

Note that it may not be possible to determine in a given program unit whether a dummy procedure is associated with a function or a subroutine. If a procedure name appears only in a dummy argument list, an EXTERNAL statement, and an actual argument list, it is not possible to determine whether the symbolic name becomes associated with a function or subroutine by examination of the subprogram alone.

15.9.3.5 Asterisks as Dummy Arguments. A dummy argument that is an asterisk is not included in the subset.

15.9.3.4 Procedures as Dummy Arguments. A dummy argument that is a dummy procedure may be associated only with an actual argument that is an intrinsic function, external function, subroutine, or another dummy procedure.

If a dummy argument is used as if it were an external function, the associated actual argument must be an intrinsic function, external function, or dummy procedure. A dummy argument that becomes associated with an intrinsic function never has any automatic typing property, even if the dummy argument name appears in Table 5 (15.10). Therefore, the type of the dummy argument must agree with the type of the result of all specific actual arguments that become associated with the dummy argument. If a dummy argument name is used as if it were an external function and that name also appears in Table 5, the intrinsic function corresponding to the dummy argument name is not available for referencing within the subprogram.

A dummy argument that is used as a procedure name in a function reference and is associated with an intrinsic function must have arguments that agree in order, number, and type with those specified in Table 5 for the intrinsic function.

If a dummy argument appears in a type-statement and an EXTERNAL statement, the actual argument must be the name of an intrinsic function, external function, or dummy procedure.

If the dummy argument is referenced as a subroutine, the actual argument must be the name of a subroutine or dummy procedure and must not appear in a type-statement or be referenced as a function.

Note that it may not be possible to determine in a given program unit whether a dummy procedure is associated with a function or a subroutine. If a procedure name appears only in a dummy argument list, an EXTERNAL statement, and an actual argument list, it is not possible to determine whether the symbolic name becomes associated with a function or subroutine by examination of the subprogram alone.

15.9.3.5 Asterisks as Dummy Arguments. A dummy argument that is an asterisk may appear only in the dummy argument list of a SUBROUTINE statement or an ENTRY statement in a subroutine subprogram.

A dummy argument that is an asterisk may be associated only with an actual argument that is an alternate return specifier in the CALL statement that identifies the current referencing name. If a dummy argument is an asterisk, the corresponding actual argument must be an alternate return specifier.

15.9.3.6 Restrictions on Association of Entities. If a subprogram reference causes a dummy argument in the referenced subprogram to become associated with another dummy argument in the referenced subprogram, neither dummy argument may become defined during execution of that subprogram. For example, if a subroutine is headed by

```
SUBROUTINE XYZ (A,B)
```

and is referenced by

```
CALL XYZ (C,C)
```

then the dummy arguments A and B each become associated with the same actual argument C and therefore with each other. Neither A nor B may become defined during this execution of subroutine XYZ or by any procedures referenced by XYZ.

If a subprogram reference causes a dummy argument to become associated with an entity in a common block in the referenced subprogram or in a subprogram referenced by the referenced subprogram, neither the dummy argument nor the entity in the common block may become defined within the subprogram or within a subprogram referenced by the referenced subprogram. For example, if a subroutine contains the statements:

```
SUBROUTINE XYZ (A)
COMMON C
```

and is referenced by a program unit that contains the statements:

```
COMMON B
CALL XYZ (B)
```

then the dummy argument A becomes associated with the actual argument B, which is associated with C, which is in a common block. Neither A nor C may become defined during execution of the subroutine XYZ or by any procedures referenced by XYZ.

15.9.4 Common Blocks. A common block provides a means of communication between external procedures or between a main program and an external procedure. The variables and arrays in a common block may be defined and referenced in all subprograms that contain a declaration of that common block. Because association is by storage rather than by name, the names of the variables and arrays may be different in the different subprograms. A reference to a datum in a common block is proper if the datum is in a defined state of the same type as the type of the name used to reference the datum. However, an integer variable that has been assigned a statement label must not be referenced in any program unit other than the one in which it was assigned (10.3).

15.9.3.6 Restrictions on Association of Entities. If a subprogram reference causes a dummy argument in the referenced subprogram to become associated with another dummy argument in the referenced subprogram, neither dummy argument may become defined during execution of that subprogram. For example, if a subroutine is headed by

```
SUBROUTINE XYZ (A,B)
```

and is referenced by

```
CALL XYZ (C,C)
```

then the dummy arguments A and B each become associated with the same actual argument C and therefore with each other. Neither A nor B may become defined during this execution of subroutine XYZ or by any procedures referenced by XYZ.

If a subprogram reference causes a dummy argument to become associated with an entity in a common block in the referenced subprogram or in a subprogram referenced by the referenced subprogram, neither the dummy argument nor the entity in the common block may become defined within the subprogram or within a subprogram referenced by the referenced subprogram. For example, if a subroutine contains the statements:

```
SUBROUTINE XYZ (A)
COMMON C
```

and is referenced by a program unit that contains the statements:

```
COMMON B
CALL XYZ (B)
```

then the dummy argument A becomes associated with the actual argument B, which is associated with C, which is in a common block. Neither A nor C may become defined during execution of the subroutine XYZ or by any procedures referenced by XYZ.

15.9.4 Common Blocks. A common block provides a means of communication between external procedures or between a main program and an external procedure. The variables and arrays in a common block may be defined and referenced in all subprograms that contain a declaration of that common block. Because association is by storage rather than by name, the names of the variables and arrays may be different in the different subprograms. A reference to a datum in a common block is proper if the datum is in a defined state of the same type as the type of the name used to reference the datum. However, an integer variable that has been assigned a statement label must not be referenced in any program unit other than the one in which it was assigned (10.3).

No difference in data type is permitted between the defined state and the type of the reference.

In a subprogram that has declared a named common block, the entities in the block remain defined after the execution of a RETURN or END statement if a common block of the same name has been declared in any program unit that is currently referencing the subprogram, either directly or indirectly. Otherwise, such entities become undefined at the execution of a RETURN or END statement, except for those that are specified by SAVE statements and those that were initially defined by DATA statements and have neither been redefined nor become undefined.

Execution of a RETURN or END statement does not cause entities in blank common or in any named common block that appears in the main program to become undefined.

Common blocks may be used also to reduce the total number of storage units required for an executable program by causing two or more subprograms to share some of the same storage units. This sharing of storage is permitted if the rules for defining and referencing data are not violated.

No difference in data type is permitted between the defined state and the type of the reference, except that either part of a complex datum may be referenced also as a real datum.

In a subprogram that has declared a named common block, the entities in the block remain defined after the execution of a RETURN or END statement if a common block of the same name has been declared in any program unit that is currently referencing the subprogram, either directly or indirectly. Otherwise, such entities become undefined at the execution of a RETURN or END statement, except for those that are specified by SAVE statements and those that were initially defined by DATA statements and have neither been redefined nor become undefined.

Execution of a RETURN or END statement does not cause entities in blank common or in any named common block that appears in the main program to become undefined.

Common blocks may be used also to reduce the total number of storage units required for an executable program by causing two or more subprograms to share some of the same storage units. This sharing of storage is permitted if the rules for defining and referencing data are not violated.

15.10 Table of Intrinsic Functions

Table 5
Intrinsic Functions

Intrinsic Function	Definition	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Function
Type Conversion	Conversion to Integer int(a) See Note 1	1		INT IFIX	Real Real	Integer Integer
	Conversion to Real See Note 2	1		REAL FLOAT	Integer Integer	Real Real
	Conversion to Double See Note 3					
	Conversion to Complex See Note 4					
	Conversion to Integer See Note 5	1		ICHAR	Character	Integer
	Conversion to Character See Note 5					
Truncation	int(a) See Note 1	1		AINT	Real	Real
Nearest Whole Number	int(a+.5) if a≥0 int(a-.5) if a<0	1		ANINT	Real	Real
Nearest Integer	int(a+.5) if a≥0 int(a-.5) if a<0	1		NINT	Real	Integer
Absolute Value	|a| See Note 6 $(ar^2+ai^2)^{1/2}$	1		IABS ABS	Integer Real	Integer Real

15.10 Table of Intrinsic Functions

Table 5
Intrinsic Functions

Intrinsic Function	Definition	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Function
Type Conversion	Conversion to Integer int(a) See Note 1	1	INT	- INT IFIX IDINT -	Integer Real Real Double Complex	Integer Integer Integer Integer Integer
	Conversion to Real See Note 2	1	REAL	REAL FLOAT - SNGL -	Integer Integer Real Double Complex	Real Real Real Real Real
	Conversion to Double See Note 3	1	DBLE	- - - -	Integer Real Double Complex	Double Double Double Double
	Conversion to Complex See Note 4	1 or 2	CMPLX	- - - -	Integer Real Double Complex	Complex Complex Complex Complex
	Conversion to Integer See Note 5	1		ICHAR	Character	Integer
	Conversion to Character See Note 5	1		CHAR	Integer	Character
Truncation	int(a) See Note 1	1	AINT	AINT DINT	Real Double	Real Double
Nearest Whole Number	int(a+.5) if a≥0 int(a-.5) if a<0	1	ANINT	ANINT DNINT	Real Double	Real Double
Nearest Integer	int(a+.5) if a≥0 int(a-.5) if a<0	1	NINT	NINT IDNINT	Real Double	Integer Integer
Absolute Value	$\|a\|$ See Note 6 $(ar^2+ai^2)^{1/2}$	1	ABS	IABS ABS DABS CABS	Integer Real Double Complex	Integer Real Double Real

Table 5 (continued)
Intrinsic Functions

Intrinsic Function	Definition	Number of Arguments	Generic Name	Specific Name	Type of	
					Argument	Function
Remaindering	$a_1 - \mathrm{int}(a_1/a_2)*a_2$ See Note 1	2		MOD AMOD	Integer Real	Integer Real
Transfer of Sign	$\lvert a_1 \rvert$ if $a_2 \geq 0$ $-\lvert a_1 \rvert$ if $a_2 < 0$	2		ISIGN SIGN	Integer Real	Integer Real
Positive Difference	$a_1 - a_2$ if $a_1 > a_2$ 0 if $a_1 \leq a_2$	2		IDIM DIM	Integer Real	Integer Real
Double Precision Product	$a_1 * a_2$					
Choosing Largest Value	$\max(a_1, a_2, \ldots)$	≥2		MAX0 AMAX1	Integer Real	Integer Real
				AMAX0 MAX1	Integer Real	Real Integer
Choosing Smallest Value	$\min(a_1, a_2, \ldots)$	≥2		MIN0 AMIN1	Integer Real	Integer Real
				AMIN0 MIN1	Integer Real	Real Integer
Length	Length of Character Entity					
Index of a Substring	Location of Substring a_2 in String a_1 See Note 10	2				
Imaginary Part of Complex Argument	ai See Note 6					
Conjugate of a Complex Argument	$(ar, -ai)$ See Note 6					
Square Root	$(a)^{1/2}$	1		SQRT	Real	Real

Table 5 (continued)
Intrinsic Functions

Intrinsic Function	Definition	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Function
Remaindering	$\underline{a}_1$-int($\underline{a}_1$/$\underline{a}_2$)*$\underline{a}_2$ See Note 1	2	MOD	MOD AMOD DMOD	Integer Real Double	Integer Real Double
Transfer of Sign	$\lvert\underline{a}_1\rvert$ if $\underline{a}_2 \geq 0$ $-\lvert\underline{a}_1\rvert$ if $\underline{a}_2 < 0$	2	SIGN	ISIGN SIGN DSIGN	Integer Real Double	Integer Real Double
Positive Difference	$\underline{a}_1-\underline{a}_2$ if $\underline{a}_1>\underline{a}_2$ 0 if $\underline{a}_1\leq\underline{a}_2$	2	DIM	IDIM DIM DDIM	Integer Real Double	Integer Real Double
Double Precision Product	$\underline{a}_1$*$\underline{a}_2$	2		DPROD	Real	Double
Choosing Largest Value	max($\underline{a}_1$,$\underline{a}_2$,...)	≥2	MAX	MAX0 AMAX1 DMAX1	Integer Real Double	Integer Real Double
				AMAX0 MAX1	Integer Real	Real Integer
Choosing Smallest Value	min($\underline{a}_1$,$\underline{a}_2$,...)	≥2	MIN	MIN0 AMIN1 DMIN1	Integer Real Double	Integer Real Double
				AMIN0 MIN1	Integer Real	Real Integer
Length	Length of Character Entity	1		LEN	Character	Integer
Index of a Substring	Location of Substring $\underline{a}_2$ in String $\underline{a}_1$ See Note 10	2		INDEX	Character	Integer
Imaginary Part of Complex Argument	ai See Note 6	1		AIMAG	Complex	Real
Conjugate of a Complex Argument	($\underline{ar}$,-$\underline{ai}$) See Note 6	1		CONJG	Complex	Complex
Square Root	$(\underline{a})^{1/2}$	1	SQRT	SQRT DSQRT CSQRT	Real Double Complex	Real Double Complex

Table 5 (continued)
Intrinsic Functions

Intrinsic Function	Definition	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Function
Exponential	e**a	1		EXP	Real	Real
Natural Logarithm	log(a)	1		ALOG	Real	Real
Common Logarithm	log10(a)	1		ALOG10	Real	Real
Sine	sin(a)	1		SIN	Real	Real
Cosine	cos(a)	1		COS	Real	Real
Tangent	tan(a)	1		TAN	Real	Real
Arcsine	arcsin(a)	1		ASIN	Real	Real
Arccosine	arccos(a)	1		ACOS	Real	Real
Arctangent	arctan(a)	1		ATAN	Real	Real
	arctan(a_1/a_2)	2		ATAN2	Real	Real
Hyperbolic Sine	sinh(a)	1		SINH	Real	Real
Hyperbolic Cosine	cosh(a)	1		COSH	Real	Real
Hyperbolic Tangent	tanh(a)	1		TANH	Real	Real

Table 5 (continued)
Intrinsic Functions

Intrinsic Function	Definition	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Function
Exponential	e**$\underline{a}$	1	EXP	EXP DEXP CEXP	Real Double Complex	Real Double Complex
Natural Logarithm	log($\underline{a}$)	1	LOG	ALOG DLOG CLOG	Real Double Complex	Real Double Complex
Common Logarithm	log10($\underline{a}$)	1	LOG10	ALOG10 DLOG10	Real Double	Real Double
Sine	sin($\underline{a}$)	1	SIN	SIN DSIN CSIN	Real Double Complex	Real Double Complex
Cosine	cos($\underline{a}$)	1	COS	COS DCOS CCOS	Real Double Complex	Real Double Complex
Tangent	tan($\underline{a}$)	1	TAN	TAN DTAN	Real Double	Real Double
Arcsine	arcsin($\underline{a}$)	1	ASIN	ASIN DASIN	Real Double	Real Double
Arccosine	arccos($\underline{a}$)	1	ACOS	ACOS DACOS	Real Double	Real Double
Arctangent	arctan($\underline{a}$)	1	ATAN	ATAN DATAN	Real Double	Real Double
	arctan($\underline{a}_1/\underline{a}_2$)	2	ATAN2	ATAN2 DATAN2	Real Double	Real Double
Hyperbolic Sine	sinh($\underline{a}$)	1	SINH	SINH DSINH	Real Double	Real Double
Hyperbolic Cosine	cosh($\underline{a}$)	1	COSH	COSH DCOSH	Real Double	Real Double
Hyperbolic Tangent	tanh($\underline{a}$)	1	TANH	TANH DTANH	Real Double	Real Double

Table 5 (continued)
Intrinsic Functions

Intrinsic Function	Definition	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Function
Lexically Greater Than or Equal	$a_1 \geq a_2$ See Note 12	2		LGE	Character	Logical
Lexically Greater Than	$a_1 > a_2$ See Note 12	2		LGT	Character	Logical
Lexically Less Than or Equal	$a_1 \leq a_2$ See Note 12	2		LLE	Character	Logical
Lexically Less Than	$a_1 < a_2$ See Note 12	2		LLT	Character	Logical

Table 5 (continued)
Intrinsic Functions

Intrinsic Function	Definition	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Function
Lexically Greater Than or Equal	$a_1 \geq a_2$ See Note 12	2		LGE	Character	Logical
Lexically Greater Than	$a_1 > a_2$ See Note 12	2		LGT	Character	Logical
Lexically Less Than or Equal	$a_1 \leq a_2$ See Note 12	2		LLE	Character	Logical
Lexically Less Than	$a_1 < a_2$ See Note 12	2		LLT	Character	Logical

Notes for Table 5:

(1) For a of type real, there are two cases: if $|a|<1$, int(a)=0; if $|a|\geq 1$, int(a) is the integer whose magnitude is the largest integer that does not exceed the magnitude of a and whose sign is the same as the sign of a. For example,

int(-3.7) = -3

For a of type real, IFIX(a) is the same as INT(a).

(2) For a of type integer, REAL(a) is as much precision of the significant part of a as a real datum can contain.

For a of type integer, FLOAT(a) is the same as REAL(a).

(3) This note does not apply to the subset.

(4) This note does not apply to the subset.

(5) ICHAR provides a means of converting from a character to an integer, based on the position of the character in the processor collating sequence. The first character in the collating sequence corresponds to position 0 and the last to position n-1, where n is the number of characters in the collating sequence.

The value of ICHAR(a) is an integer in the range: $0 \leq$ ICHAR(a) $\leq n-1$, where a is an argument of type character of length one. The value of a must be a

Notes for Table 5:

(1) For a of type integer, int(a)=a. For a of type real or double precision, there are two cases: if $|a| < 1$, int(a)=0; if $|a| \geq 1$, int(a) is the integer whose magnitude is the largest integer that does not exceed the magnitude of a and whose sign is the same as the sign of a. For example,

int(-3.7) = -3

For a of type complex, int(a) is the value obtained by applying the above rule to the real part of a.

For a of type real, IFIX(a) is the same as INT(a).

(2) For a of type real, REAL(a) is a. For a of type integer or double precision, REAL(a) is as much precision of the significant part of a as a real datum can contain. For a of type complex, REAL(a) is the real part of a.

For a of type integer, FLOAT(a) is the same as REAL(a).

(3) For a of type double precision, DBLE(a) is a. For a of type integer or real, DBLE(a) is as much precision of the significant part of a as a double precision datum can contain. For a of type complex, DBLE(a) is as much precision of the significant part of the real part of a as a double precision datum can contain.

(4) CMPLX may have one or two arguments. If there is one argument, it may be of type integer, real, double precision, or complex. If there are two arguments, they must both be of the same type and may be of type integer, real, or double precision.

For a of type complex, CMPLX(a) is a. For a of type integer, real, or double precision, CMPLX(a) is the complex value whose real part is REAL(a) and whose imaginary part is zero.

CMPLX(a_1,a_2) is the complex value whose real part is REAL(a_1) and whose imaginary part is REAL(a_2).

(5) ICHAR provides a means of converting from a character to an integer, based on the position of the character in the processor collating sequence. The first character in the collating sequence corresponds to position 0 and the last to position n-1, where n is the number of characters in the collating sequence.

The value of ICHAR(a) is an integer in the range: $0 \leq$ ICHAR(a) $\leq n-1$, where a is an argument of type character of length one. The value of a must be a character capable of representation in the processor. The position of that character in the collating sequence is the value of ICHAR.

For any characters $\underline{c}_1$ and $\underline{c}_2$ capable of representation in the processor, ($\underline{c}_1$ .LE. $\underline{c}_2$) is true if and only if (ICHAR($\underline{c}_1$) .LE. ICHAR($\underline{c}_2$)) is true, and ($\underline{c}_1$ .EQ. $\underline{c}_2$) is true if and only if (ICHAR($\underline{c}_1$) .EQ. ICHAR($\underline{c}_2$)) is true.

The CHAR function is not included in the subset.

(6) This note does not apply to the subset.

(7) All angles are expressed in radians.

(8) This note does not apply to the subset.

(9) All arguments in an intrinsic function reference must be of the same type.

(10) The INDEX function is not included in the subset.

(11) There are some names in Table 5 of the full language that must not be used as intrinsic function names in a standard-conforming program at the subset level. If such a name is used as an external function name, the name must appear in an EXTERNAL statement in each program unit that references the external function. The only names in Table 5 that may be used as specific names of intrinsic functions are the following:

ABS	AMIN0	COS	IDIM	LLT	REAL
ACOS	AMIN1	COSH	IFIX	MAX0	SIGN
AINT	AMOD	DIM	INT	MAX1	SIN
ALOG	ANINT	EXP	ISIGN	MOD	SINH
ALOG10	ASIN	FLOAT	LGE	MIN0	SQRT
AMAX0	ATAN	IABS	LGT	MIN1	TAN
AMAX1	ATAN2	ICHAR	LLE	NINT	TANH

character capable of representation in the processor. The position of that character in the collating sequence is the value of ICHAR.

For any characters c_1 and c_2 capable of representation in the processor, (c_1 .LE. c_2) is true if and only if (ICHAR(c_1) .LE. ICHAR(c_2)) is true, and (c_1 .EQ. c_2) is true if and only if (ICHAR(c_1) .EQ. ICHAR(c_2)) is true.

CHAR(i) returns the character in the ith position of the processor collating sequence. The value is of type character of length one. i must be an integer expression whose value must be in the range $0 \le i \le n-1$.

ICHAR(CHAR(i)) = i for $0 \le i \le n-1$.

CHAR(ICHAR(c)) = c for any character c capable of representation in the processor.

(6) A complex value is expressed as an ordered pair of reals, (ar,ai), where ar is the real part and ai is the imaginary part.

(7) All angles are expressed in radians.

(8) The result of a function of type complex is the principal value.

(9) All arguments in an intrinsic function reference must be of the same type.

(10) INDEX(a_1,a_2) returns an integer value indicating the starting position within the character string a_1 of a substring identical to string a_2. If a_2 occurs more than once in a_1, the starting postion of the first occurence is returned.

If a_2 does not occur in a_1, the value zero is returned. Note that zero is returned if LEN(a_1) < LEN(a_2).

(11) The value of the argument of the LEN function need not be defined at the time the function reference is executed.

(12) LGE($\underline{a}_1$,$\underline{a}_2$) returns the value true if $\underline{a}_1$=$\underline{a}_2$ or if $\underline{a}_1$ follows $\underline{a}_2$ in the collating sequence described in American National Standard Code for Information Interchange, ANSI X3.4-1977 (ASCII), and otherwise returns the value false.

LGT($\underline{a}_1$,$\underline{a}_2$) returns the value true if $\underline{a}_1$ follows $\underline{a}_2$ in the collating sequence described in ANSI X3.4-1977 (ASCII), and otherwise returns the value false.

LLE($\underline{a}_1$,$\underline{a}_2$) returns the value true if $\underline{a}_1$=$\underline{a}_2$ or if $\underline{a}_1$ precedes $\underline{a}_2$ in the collating sequence described in ANSI X3.4-1977 (ASCII), and otherwise returns the value false.

LLT($\underline{a}_1$,$\underline{a}_2$) returns the value true if $\underline{a}_1$ precedes $\underline{a}_2$ in the collating sequence described in ANSI X3.4-1977 (ASCII), and otherwise returns the value false.

The operands for LGE, LGT, LLE, and LLT must be of the same length.

If either of the character entities being compared contains a character that is not in the ASCII character set, the result is processor-dependent.

15.10.1 <u>Restrictions on Range of Arguments and Results</u>. Restrictions on the range of arguments and results for intrinsic functions are as follows:

(1) Remaindering: The result for MOD and AMOD is undefined when the value of the second argument is zero.

(2) Transfer of Sign: If the value of the first argument of ISIGN or SIGN is zero, the result is zero, which is neither positive or negative (4.1.3).

(3) Square Root: The value of the argument of SQRT must be greater than or equal to zero.

(4) Logarithms: The value of the argument of ALOG and ALOG10 must be greater than zero.

(12) LGE(a_1,a_2) returns the value true if a_1=a_2 or if a_1 follows a_2 in the collating sequence described in American National Standard Code for Information Interchange, ANSI X3.4-1977 (ASCII), and otherwise returns the value false.

LGT(a_1,a_2) returns the value true if a_1 follows a_2 in the collating sequence described in ANSI X3.4-1977 (ASCII), and otherwise returns the value false.

LLE(a_1,a_2) returns the value true if a_1=a_2 or if a_1 precedes a_2 in the collating sequence described in ANSI X3.4-1977 (ASCII), and otherwise returns the value false.

LLT(a_1,a_2) returns the value true if a_1 precedes a_2 in the collating sequence described in ANSI X3.4-1977 (ASCII), and otherwise returns the value false.

If the operands for LGE, LGT, LLE, and LLT are of unequal length, the shorter operand is considered as if it were extended on the right with blanks to the length of the longer operand.

If either of the character entities being compared contains a character that is not in the ASCII character set, the result is processor-dependent.

15.10.1 Restrictions on Range of Arguments and Results. Restrictions on the range of arguments and results for intrinsic functions when referenced by their specific names are as follows:

(1) Remaindering: The result for MOD, AMOD, and DMOD is undefined when the value of the second argument is zero.

(2) Transfer of Sign: If the value of the first argument of ISIGN, SIGN, or DSIGN is zero, the result is zero, which is neither positive or negative (4.1.3).

(3) Square Root: The value of the argument of SQRT and DSQRT must be greater than or equal to zero. The result of CSQRT is the principal value with the real part greater than or equal to zero. When the real part of the result is zero, the imaginary part is greater than or equal to zero.

(4) Logarithms: The value of the argument of ALOG, DLOG, ALOG10, and DLOG10 must be greater than zero. The value of the argument of CLOG must not be (0.,0.). The range of the imaginary part of the result of CLOG is: $-\pi <$ imaginary part $\leq \pi$. The imaginary part of the result is π only when the real part of the argument is less than zero and the imaginary part of the argument is zero.

(5) Sine, Cosine, and Tangent: The absolute value of the argument of SIN, COS, and TAN is not restricted to be less than 2π.

(6) Arcsine: The absolute value of the argument of ASIN must be less than or equal to one. The range of the result is: $-\pi/2 \le$ result $\le \pi/2$.

(7) Arccosine: The absolute value of the argument of ACOS must be less than or equal to one. The range of the result is: $0 \le$ result $\le \pi$.

(8) Arctangent: The range of the result for ATAN is: $-\pi/2 \le$ result $\le \pi/2$. If the value of the first argument of ATAN2 is positive, the result is positive. If the value of the first argument is zero, the result is zero if the second argument is positive and π if the second argument is negative. If the value of the first argument is negative, the result is negative. If the value of the second argument is zero, the absolute value of the result is $\pi/2$. The arguments must not both have the value zero. The range of the result for ATAN2 is: $-\pi <$ result $\le \pi$.

(5) Sine, Cosine, and Tangent: The absolute value of the argument of SIN, DSIN, COS, DCOS, TAN, and DTAN is not restricted to be less than 2π.

(6) Arcsine: The absolute value of the argument of ASIN and DASIN must be less than or equal to one. The range of the result is: $-\pi/2 \le \text{result} \le \pi/2$.

(7) Arccosine: The absolute value of the argument of ACOS and DACOS must be less than or equal to one. The range of the result is: $0 \le \text{result} \le \pi$.

(8) Arctangent: The range of the result for ATAN and DATAN is: $-\pi/2 \le \text{result} \le \pi/2$. If the value of the first argument of ATAN2 or DATAN2 is positive, the result is positive. If the value of the first argument is zero, the result is zero if the second argument is positive and π if the second argument is negative. If the value of the first argument is negative, the result is negative. If the value of the second argument is zero, the absolute value of the result is $\pi/2$. The arguments must not both have the value zero. The range of the result for ATAN2 and DATAN2 is: $-\pi < \text{result} \le \pi$.

The above restrictions on arguments and results also apply to the intrinsic functions when referenced by their generic names.

16. BLOCK DATA SUBPROGRAM

Block data subprograms are not included in the subset.

16.1 BLOCK DATA Statement

The BLOCK DATA statement is not included in the subset.

16.2 Block Data Subprogram Restrictions

Block data subprograms are not included in the subset.

16. BLOCK DATA SUBPROGRAM

Block data subprograms are used to provide initial values for variables and array elements in named common blocks.

A block data subprogram is a program unit that has a BLOCK DATA statement as its first statement. A block data subprogram is nonexecutable. There may be more than one block data subprogram in an executable program.

16.1 BLOCK DATA Statement

The form of a BLOCK DATA statement is:

BLOCK DATA [sub]

where sub is the symbolic name of the block data subprogram in which the BLOCK DATA statement appears.

The optional name sub is a global name (18.1.1) and must not be the same as the name of an external procedure, main program, common block, or other block data subprogram in the same executable program. The name sub must not be the same as any local name in the subprogram.

16.2 Block Data Subprogram Restrictions

The BLOCK DATA statement must appear only as the first statement of a block data subprogram. The only other statements that may appear in a block data subprogram are IMPLICIT, PARAMETER, DIMENSION, COMMON, SAVE, EQUIVALENCE, DATA, END, and type-statements. Note that comment lines are permitted.

If an entity in a named common block is initially defined, all entities having storage units in the common block storage sequence must be specified even if they are not all initially defined. More than one named common block may have entities initially defined in a single block data subprogram.

Only an entity in a named common block may be initially defined in a block data subprogram. Note that entities associated with an entity in a common block are considered to be in that common block.

The same named common block may not be specified in more than one block data subprogram in the same executable program.

There must not be more than one unnamed block data subprogram in an executable program.

17. ASSOCIATION AND DEFINITION

17.1 Storage and Association

Storage sequences are used to describe relationships that exist among variables, array elements, common blocks, and arguments.

17.1.1 Storage Sequence. A storage sequence is a sequence (2.1) of storage units. The size of a storage sequence is the number of storage units in the storage sequence. A storage unit is a character storage unit or a numeric storage unit.

A variable or array element of type integer, real, or logical has a storage sequence of one numeric storage unit.

A variable or array element of type character has a storage sequence of character storage units. The number of character storage units in the storage sequence is the length of the character entity. The order of the sequence corresponds to the ordering of character positions (4.8).

Each array and common block has a storage sequence (5.2.5 and 8.3.2).

17.1.2 Association of Storage Sequences. Two storage sequences s_1 and s_2 are associated if the ith storage unit of s_1 is the same as the jth storage unit of s_2. This causes the (i+k)th storage unit of s_1 to be the same as the (j+k)th storage unit of s_2, for each integer k such that $1 \leq i+k \leq$ size of s_1 and $1 \leq j+k \leq$ size of s_2.

17.1.3 Association of Entities. Two variables or array elements are associated if their storage sequences are associated. Two entities are totally associated if they have the same storage sequence. Partial association of character entities is prohibited.

The definition status and value of an entity affects the definition status and value of any associated entity. An EQUIVALENCE statement, a COMMON statement, or a procedure reference (argument association) may cause association of storage sequences.

An EQUIVALENCE statement causes association of entities only within one program unit, unless one of the equivalenced entities is also in a common block (8.3).

17. ASSOCIATION AND DEFINITION

17.1 Storage and Association

Storage sequences are used to describe relationships that exist among variables, array elements, substrings, common blocks, and arguments.

17.1.1 Storage Sequence. A storage sequence is a sequence (2.1) of storage units. The size of a storage sequence is the number of storage units in the storage sequence. A storage unit is a character storage unit or a numeric storage unit.

A variable or array element of type integer, real, or logical has a storage sequence of one numeric storage unit.

A variable or array element of type double precision or complex has a storage sequence of two numeric storage units. In a complex storage sequence, the real part has the first storage unit and the imaginary part has the second storage unit.

A variable, array element, or substring of type character has a storage sequence of character storage units. The number of character storage units in the storage sequence is the length of the character entity. The order of the sequence corresponds to the ordering of character positions (4.8).

Each array and common block has a storage sequence (5.2.5 and 8.3.2).

17.1.2 Association of Storage Sequences. Two storage sequences s_1 and s_2 are associated if the ith storage unit of s_1 is the same as the jth storage unit of s_2. This causes the (i+k)th storage unit of s_1 to be the same as the (j+k)th storage unit of s_2, for each integer k such that $1 \le i+k \le$ size of s_1 and $1 \le j+k \le$ size of s_2.

17.1.3 Association of Entities. Two variables, array elements, or substrings are associated if their storage sequences are associated. Two entities are totally associated if they have the same storage sequence. Two entities are partially associated if they are associated but not totally associated.

The definition status and value of an entity affects the definition status and value of any associated entity. An EQUIVALENCE statement, a COMMON statement, an ENTRY statement (15.7.3), or a procedure reference (argument association) may cause association of storage sequences.

An EQUIVALENCE statement causes association of entities only within one program unit, unless one of the equivalenced entities is also in a common block (8.3).

Arguments and COMMON statements cause entities in one program unit to become associated with entities in another program unit (8.3 and 15.9). Note that association between actual and dummy arguments does not imply association of storage sequences except when the actual argument is the name of a variable, array element, or array.

In the example:

```
REAL A(4),B
EQUIVALENCE (A(2),B)
```

the second storage unit of A and the storage unit of B are specified as the same. The storage sequences may be illustrated as:

```
storage unit      |  1   |  2   |  3   |  4   |
                  | A(1) | A(2) | A(3) | A(4) |
                         |--B-- |
```

A(2) and B are totally associated.

Arguments and COMMON statements cause entities in one program unit to become associated with entities in another program unit (8.3 and 15.9). Note that association between actual and dummy arguments does not imply association of storage sequences except when the actual argument is the name of a variable, array element, array, or substring.

In a function subprogram, an ENTRY statement causes the entry name to become associated with the name of the function subprogram which appears in the FUNCTION statement.

Partial association may exist only between two character entities or between a double precision or complex entity and an entity of type integer, real, logical, double precision, or complex.

Except for character entities, partial association may occur only through the use of COMMON, EQUIVALENCE, or ENTRY statements. Partial association must not occur through argument association, except for arguments of type character.

In the example:

```
      REAL A(4),B
      COMPLEX C(2)
      DOUBLE PRECISION D
      EQUIVALENCE (C(2),A(2),B), (A,D)
```

the third storage unit of C, the second storage unit of A, the storage unit of B, and the second storage unit of D are specified as the same. The storage sequences may be illustrated as:

```
storage unit    |   1   |  2   |   3  |  4   |   5  |
                |----C(1)---|----C(2)---|
                        | A(1) | A(2) | A(3) | A(4) |
                               |--B-- |
                        |-----D------|
```

A(2) and B are totally associated. The following are partially associated: A(1) and C(1), A(2) and C(2), A(3) and C(2), B and C(2), A(1) and D, A(2) and D, B and D, C(1) and D, and C(2) and D. Note that although C(1) and C(2) are each associated with D, C(1) and C(2) are not associated with each other.

Partial association of character entities occurs when some, but not all, of the storage units of the entities are the same. In the example:

```
      CHARACTER A*4,B*4,C*3
      EQUIVALENCE (A(2:3),B,C)
```

A, B, and C are partially associated.

17.2 Events That Cause Entities to Become Defined

Variables and array elements become defined as follows:

(1) Execution of an arithmetic, logical, or character assignment statement causes the entity that precedes the equals to become defined.

(2) As execution of an input statement proceeds, each entity that is assigned a value of its corresponding type from the input medium becomes defined at the time of such assignment.

(3) Execution of a DO statement causes the DO-variable to become defined.

(4) Beginning of execution of action specified by an implied-DO list in an input/output statement causes the implied-DO-variable to become defined.

(5) A DATA statement causes entities to become initially defined at the beginning of execution of an executable program.

(6) Execution of an ASSIGN statement causes the variable in the statement to become defined with a statement label value.

(7) When an entity of a given type becomes defined, all totally associated entities of the same type become defined except that entities totally associated with the variable in an ASSIGN statement become undefined when the ASSIGN statement is executed.

(8) A reference to a subprogram causes a dummy argument to become defined if the corresponding actual argument is defined with a value that is not a statement label value. Note that there must be agreement between the actual argument and the dummy argument (15.9.3).

17.2 Events That Cause Entities to Become Defined

Variables, array elements, and substrings become defined as follows:

(1) Execution of an arithmetic, logical, or character assignment statement causes the entity that precedes the equals to become defined.

(2) As execution of an input statement proceeds, each entity that is assigned a value of its corresponding type from the input medium becomes defined at the time of such assignment.

(3) Execution of a DO statement causes the DO-variable to become defined.

(4) Beginning of execution of action specified by an implied-DO list in an input/output statement causes the implied-DO-variable to become defined.

(5) A DATA statement causes entities to become initially defined at the beginning of execution of an executable program.

(6) Execution of an ASSIGN statement causes the variable in the statement to become defined with a statement label value.

(7) When an entity of a given type becomes defined, all totally associated entities of the same type become defined except that entities totally associated with the variable in an ASSIGN statement become undefined when the ASSIGN statement is executed.

(8) A reference to a subprogram causes a dummy argument to become defined if the corresponding actual argument is defined with a value that is not a statement label value. Note that there must be agreement between the actual argument and the dummy argument (15.9.3).

(9) Execution of an input/output statement containing an input/output status specifier causes the specified integer variable or array element to become defined.

(10) Execution of an INQUIRE statement causes any entity that is assigned a value during the execution of the statement to become defined if no error condition exists.

(11) When a complex entity becomes defined, all partially associated real entities become defined.

(12) When both parts of a complex entity become defined as a result of partially associated real or complex

17.3 Events That Cause Entities to Become Undefined

Variables and array elements become undefined as follows:

(1) All entities are undefined at the beginning of execution of an executable program except those entities initially defined by DATA statements.

(2) When an entity of a given type becomes defined, all totally associated entities of different type become undefined.

(3) Execution of an ASSIGN statement causes the variable in the statement to become undefined as an integer. Entities that are associated with the variable become undefined.

(4) When the evaluation of a function causes an argument of the function or an entity in common to become defined and if a reference to the function appears in an expression in which the value of the function is not needed to determine the value of the expression, then the argument or the entity in common becomes undefined when the expression is evaluated (6.6.1).

(5) The execution of a RETURN statement or an END statement within a subprogram causes all entities within the subprogram to become undefined except for the following:

   (a) Entities in blank common

   (b) Initially defined entities that have neither been redefined nor become undefined

   (c) Entities specified by SAVE statements entities becoming defined, the complex entity becomes defined.

(13) When all characters of a character entity become defined, the character entity becomes defined.

17.3 Events That Cause Entities to Become Undefined

Variables, array elements, and substrings become undefined as follows:

(1) All entities are undefined at the beginning of execution of an executable program except those entities initially defined by DATA statements.

(2) When an entity of a given type becomes defined, all totally associated entities of different type become undefined.

(3) Execution of an ASSIGN statement causes the variable in the statement to become undefined as an integer. Entities that are associated with the variable become undefined.

(4) When an entity of type other than character becomes defined, all partially associated entities become undefined. However, when an entity of type real is partially associated with an entity of type complex, the complex entity does not become undefined when the real entity becomes defined and the real entity does not become undefined when the complex entity becomes defined. When an entity of type complex is partially associated with another entity of type complex, definition of one entity does not cause the other to become undefined.

(5) When the evaluation of a function causes an argument of the function or an entity in common to become defined and if a reference to the function appears in an expression in which the value of the function is not needed to determine the value of the expression, then the argument or the entity in common becomes undefined when the expression is evaluated (6.6.1).

(6) The execution of a RETURN statement or an END statement within a subprogram causes all entities within the subprogram to become undefined except for the following:

(a) Entities in blank common

(b) Initially defined entities that have neither been redefined nor become undefined

(c) Entities specified by SAVE statements

(d) Entities in a named common block that appears in the subprogram and appears in at least one other program unit that is either directly or indirectly referencing the subprogram

(6) When an end-of-file condition occurs during execution of an input statement, all of the entities specified by the input list of the statement become undefined.

(7) Execution of a direct access input statement that specifies a record that has not been previously written causes all of the entities specified by the input list of the statement to become undefined.

(8) When an entity becomes undefined as a result of conditions described in (4) through (7), all totally associated entities become undefined.

(d) Entities in a named common block that appears in the subprogram and appears in at least one other program unit that is either directly or indirectly referencing the subprogram

(7) When an error condition or end-of-file condition occurs during execution of an input statement, all of the entities specified by the input list of the statement become undefined.

(8) Execution of a direct access input statement that specifies a record that has not been previously written causes all of the entities specified by the input list of the statement to become undefined.

(9) Execution of an INQUIRE statement may cause entities to become undefined (12.10.3).

(10) When any character of a character entity becomes undefined, the character entity becomes undefined.

(11) When an entity becomes undefined as a result of conditions described in (5) through (10), all totally associated entities become undefined and all partially associated entities of type other than character become undefined.

18. SCOPE AND CLASSES OF SYMBOLIC NAMES

A symbolic name consists of one to six alphanumeric characters, the first of which must be a letter. Some sequences of characters, such as format edit descriptors and keywords that uniquely identify certain statements, for example, GO TO, READ, FORMAT, etc, are not symbolic names in such occurrences nor do they form the first characters of symbolic names in such occurrences.

18.1 Scope of Symbolic Names

The scope of a symbolic name is an executable program, a program unit, or a statement function statement.

The name of the main program and the names of external functions, subroutines, and common blocks have a scope of an executable program.

The names of variables, arrays, statement functions, intrinsic functions, and dummy procedures have a scope of a program unit.

The names of variables that appear as dummy arguments in a statement function statement have a scope of that statement.

18.1.1 Global Entities. The main program, common blocks, subprograms, and external procedures are global entities of an executable program. A symbolic name that identifies a global entity must not be used to identify any other global entity in the same executable program.

18.1.1.1 Classes of Global Entities. A symbolic name in one of the following classes is a global entity in an executable program:

(1) Common block

(2) External function

(3) Subroutine

(4) Main program

18.1.2 Local Entities. The symbolic name of a local entity identifies that entity in a single program unit. Within a program unit, a symbolic name that is in one class of entities local to the program unit must not also be in another class of entities local to the program unit.

18. SCOPE AND CLASSES OF SYMBOLIC NAMES

A symbolic name consists of one to six alphanumeric characters, the first of which must be a letter. Some sequences of characters, such as format edit descriptors and keywords that uniquely identify certain statements, for example, GO TO, READ, FORMAT, etc, are not symbolic names in such occurrences nor do they form the first characters of symbolic names in such occurrences.

18.1 Scope of Symbolic Names

The scope of a symbolic name is an executable program, a program unit, a statement function statement, or an implied-DO list in a DATA statement.

The name of the main program and the names of block data subprograms, external functions, subroutines, and common blocks have a scope of an executable program.

The names of variables, arrays, constants, statement functions, intrinsic functions, and dummy procedures have a scope of a program unit.

The names of variables that appear as dummy arguments in a statement function statement have a scope of that statement.

The names of variables that appear as the DO-variable of an implied-DO in a DATA statement have a scope of the implied-DO list.

18.1.1 Global Entities. The main program, common blocks, subprograms, and external procedures are global entities of an executable program. A symbolic name that identifies a global entity must not be used to identify any other global entity in the same executable program.

18.1.1.1 Classes of Global Entities. A symbolic name in one of the following classes is a global entity in an executable program:

(1) Common block

(2) External function

(3) Subroutine

(4) Main program

(5) Block data subprogram

18.1.2 Local Entities. The symbolic name of a local entity identifies that entity in a single program unit. Within a program unit, a symbolic name that is in one class of entities local to the program unit must not also be in another class of entities local to the program unit.

However, a symbolic name that identifies a local entity may, in a different program unit, identify an entity of any class that is either local to that program unit or global to the executable program. A symbolic name that identifies a global entity in a program unit must not be used to identify a local entity in that program unit, except for a common block name and an external function name (18.2.1 and 18.2.2).

18.1.2.1 Classes of Local Entities. A symbolic name in one of the following classes is a local entity in a program unit.

(1) Array

(2) Variable

(3) Statement function

(4) Intrinsic function

(5) Dummy procedure

A symbolic name that is a dummy argument of a procedure is classified as a variable, array, or dummy procedure. The specification and usage must not violate the respective class rules.

18.2 Classes of Symbolic Names

In a program unit, a symbolic name must not be in more than one class except as noted in the following paragraphs of this section. There are no restrictions on the appearances of the same symbolic name in different program units of an executable program other than those noted in this section.

18.2.1 Common Block. A symbolic name is the name of a common block if and only if it appears as a block name in a COMMON statement (8.3).

A common block name is global to the executable program.

A common block name in a program unit may also be the name of any local entity other than an intrinsic function or a local variable that is also an external function in a function subprogram. If a name is used for both a common block and a local entity, the appearance of that name in any context other than as a common block name in a COMMON or SAVE statement identifies only the local entity. Note that an intrinsic function name may be a common block name in a program unit that does not reference the intrinsic function.

However, a symbolic name that identifies a local entity may, in a different program unit, identify an entity of any class that is either local to that program unit or global to the executable program. A symbolic name that identifies a global entity in a program unit must not be used to identify a local entity in that program unit, except for a common block name and an external function name (18.2.1 and 18.2.2).

18.1.2.1 Classes of Local Entities. A symbolic name in one of the following classes is a local entity in a program unit.

(1) Array

(2) Variable

(3) Constant

(4) Statement function

(5) Intrinsic function

(6) Dummy procedure

A symbolic name that is a dummy argument of a procedure is classified as a variable, array, or dummy procedure. The specification and usage must not violate the respective class rules.

18.2 Classes of Symbolic Names

In a program unit, a symbolic name must not be in more than one class except as noted in the following paragraphs of this section. There are no restrictions on the appearances of the same symbolic name in different program units of an executable program other than those noted in this section.

18.2.1 Common Block. A symbolic name is the name of a common block if and only if it appears as a block name in a COMMON statement (8.3).

A common block name is global to the executable program.

A common block name in a program unit may also be the name of any local entity other than a constant, intrinsic function, or a local variable that is also an external function in a function subprogram. If a name is used for both a common block and a local entity, the appearance of that name in any context other than as a common block name in a COMMON or SAVE statement identifies only the local entity. Note that an intrinsic function name may be a common block name in a program unit that does not reference the intrinsic function.

18.2.2 External Function. A symbolic name is the name of an external function if it meets either of the following conditions:

(1) The name appears immediately following the word FUNCTION in a FUNCTION statement.

(2) It is not an array name, character variable name, statement function name, intrinsic function name, dummy argument, or subroutine name, and every appearance is immediately followed by a left parenthesis except in a type-statement, in an EXTERNAL statement, or as an actual argument.

In a function subprogram, the name of a function that appears immediately after the word FUNCTION in a FUNCTION statement must also be the name of a variable in that subprogram (15.5.1).

An external function name is global to the executable program.

18.2.3 Subroutine. A symbolic name is the name of a subroutine if it meets either of the following conditions:

(1) The name appears immediately following the word SUBROUTINE in a SUBROUTINE statement.

(2) The name appears immediately following the word CALL in a CALL statement and is not a dummy argument.

A subroutine name is global to the executable program.

18.2.4 Main Program. A symbolic name is the name of a main program if and only if it appears in a PROGRAM statement in the main program.

A main program name is global to the executable program.

18.2.5 Block Data Subprogram. Block data subprograms are not included in the subset.

18.2.6 Array. A symbolic name is the name of an array if it appears as the array name in an array declarator (5.1) in a DIMENSION, COMMON, or type-statement.

An array name is local to a program unit.

18.2.2 External Function. A symbolic name is the name of an external function if it meets either of the following conditions:

(1) The name appears immediately following the word FUNCTION in a FUNCTION statement or the word ENTRY in an ENTRY statement within a function subprogram.

(2) It is not an array name, character variable name, statement function name, intrinsic function name, dummy argument, or subroutine name, and every appearance is immediately followed by a left parenthesis except in a type-statement, in an EXTERNAL statement, or as an actual argument.

In a function subprogram, the name of a function that appears immediately after the word FUNCTION in a FUNCTION statement or immediately after the word ENTRY in an ENTRY statement may also be the name of a variable in that subprogram (15.5.1). At least one such function name must be the name of a variable in a function subprogram.

An external function name is global to the executable program.

18.2.3 Subroutine. A symbolic name is the name of a subroutine if it meets either of the following conditions:

(1) The name appears immediately following the word SUBROUTINE in a SUBROUTINE statement or the word ENTRY in an ENTRY statement within a subroutine subprogram.

(2) The name appears immediately following the word CALL in a CALL statement and is not a dummy argument.

A subroutine name is global to the executable program.

18.2.4 Main Program. A symbolic name is the name of a main program if and only if it appears in a PROGRAM statement in the main program.

A main program name is global to the executable program.

18.2.5 Block Data Subprogram. A symbolic name is the name of a block data subprogram if and only if it appears in a BLOCK DATA statement.

A block data subprogram name is global to the executable program.

18.2.6 Array. A symbolic name is the name of an array if it appears as the array name in an array declarator (5.1) in a DIMENSION, COMMON, or type-statement.

An array name is local to a program unit.

An array name may be the same as a common block name.

18.2.7 Variable. A symbolic name is the name of a variable if it meets all of the following conditions:

(1) It does not appear in an INTRINSIC or EXTERNAL statement.

(2) It is not the name of an array, subroutine, or main program.

(3) It appears other than as the name of a common block or the name of an external function in a FUNCTION statement.

(4) It is never immediately followed by a left parenthesis unless it is immediately preceded by the word FUNCTION in a FUNCTION statement.

A variable name in the dummy argument list of a statement function statement is local to the statement function statement in which it occurs. Note that the use of a name that appears in Table 5 as a dummy argument of a statement function removes it from the class of intrinsic functions. All other variable names are local to a program unit.

A statement function dummy argument name may also be the name of a variable or common block in the program unit. The appearance of the name in any context other than as a dummy argument of the statement function identifies the local variable or common block. The statement function dummy argument name and local variable name have the same type and, if of type character, both have the same constant length.

18.2.8 Constant. • Symbolic names of constants are not included in the subset.

18.2.9 Statement Function. A symbolic name is the name of a statement function if a statement function statement

An array name may be the same as a common block name.

18.2.7 Variable. A symbolic name is the name of a variable if it meets all of the following conditions:

(1) It does not appear in a PARAMETER, INTRINSIC, or EXTERNAL statement.

(2) It is not the name of an array, subroutine, main program, or block data subprogram.

(3) It appears other than as the name of a common block, the name of an external function in a FUNCTION statement, or an entry name in an ENTRY statement in an external function.

(4) It is never immediately followed by a left parenthesis unless it is immediately preceded by the word FUNCTION in a FUNCTION statement, is immediately preceded by the word ENTRY in an ENTRY statement, or is at the beginning of a character substring name (5.7.1).

A variable name in the dummy argument list of a statement function statement is local to the statement function statement in which it occurs. Note that the use of a name that appears in Table 5 as a dummy argument of a statement function removes it from the class of intrinsic functions. A variable name that appears as an implied-DO-variable in a DATA statement is local to the implied-DO list. All other variable names are local to a program unit,

A statement function dummy argument name may also be the name of a variable or common block in the program unit. The appearance of the name in any context other than as a dummy argument of the statement function identifies the local variable or common block. The statement function dummy argument name and local variable name have the same type and, if of type character, both have the same constant length.

The name of an implied-DO-variable in a DATA statement may also be the name of a variable or common block in the program unit. The appearance of the name in any context other than as an implied-DO-variable in the DATA statement identifies the local variable or common block. The implied-DO-variable and the local variable have the same type.

18.2.8 Constant. A symbolic name is the name of a constant if it appears as a symbolic name in a PARAMETER statement.

The symbolic name of a constant is local to a program unit.

18.2.9 Statement Function. A symbolic name is the name of a statement function if a statement function statement

(15.4) is present for that symbolic name and it is not an array name.

A statement function name is local to a program unit. A statement function name may be the same as a common block name.

18.2.10 Intrinsic Function. A symbolic name is the name of an intrinsic function if it meets all of the following conditions:

(1) The name appears in the Specific Name column of Table 5 and in the list of subset intrinsic functions in Note 11 of Table 5.

(2) It is not an array name, statement function name, subroutine name, or dummy argument name.

(3) Every appearance of the symbolic name, except in an INTRINSIC statement, a type-statement, or as an actual argument, is immediately followed by an actual argument list enclosed in parentheses.

An intrinsic function name is local to a program unit.

18.2.11 Dummy Procedure. A symbolic name is the name of a dummy procedure if the name appears in the dummy argument list of a FUNCTION or SUBROUTINE statement and meets one or more of the following conditions:

(1) It appears in an EXTERNAL statement.

(2) It appears immediately following the word CALL in a CALL statement.

(3) It is not an array name or character variable name, and every appearance is immediately followed by a left parenthesis except in a type-statement, in an EXTERNAL statement, in a CALL statement, as a dummy argument, as an actual argument, or as a common block name in a COMMON or SAVE statement.

A dummy procedure name is local to a program unit.

A dummy procedure must not be of type character.

(15.4) is present for that symbolic name and it is not an array name.

A statement function name is local to a program unit. A statement function name may be the same as a common block name.

18.2.10 Intrinsic Function. A symbolic name is the name of an intrinsic function if it meets all of the following conditions:

(1) The name appears in the Specific Name column or the Generic Name column of Table 5.

(2) It is not an array name, statement function name, subroutine name, or dummy argument name.

(3) Every appearance of the symbolic name, except in an INTRINSIC statement, a type-statement, or as an actual argument, is immediately followed by an actual argument list enclosed in parentheses.

An intrinsic function name is local to a program unit.

18.2.11 Dummy Procedure. A symbolic name is the name of a dummy procedure if the name appears in the dummy argument list of a FUNCTION, SUBROUTINE, or ENTRY statement and meets one or more of the following conditions:

(1) It appears in an EXTERNAL statement.

(2) It appears immediately following the word CALL in a CALL statement.

(3) It is not an array name or character variable name, and every appearance is immediately followed by a left parenthesis except in a type-statement, in an EXTERNAL statement, in a CALL statement, as a dummy argument, as an actual argument, or as a common block name in a COMMON or SAVE statement.

A dummy procedure name is local to a program unit.

APPENDIXES

(These Appendixes are not part of American National Standard Programming Language FORTRAN, ANSI X3.9-1978, but are included for information purposes only.)

APPENDIX A: CRITERIA, CONFLICTS, AND PORTABILITY

A1 Criteria

The principal criteria used in developing this FORTRAN standard were:

(1) Interchangeability of FORTRAN programs between processors

(2) Compatibility with ANSI X3.9-1966, allied standards, and existing practices

(3) Consistency and simplicity to user

(4) Suitability for efficient processor operation for a wide range of computing equipment of varying structure and power

(5) Allowance for future growth in the language

(6) Achievement of capabilities not currently available, but needed for processes appropriately expressed in FORTRAN

(7) Acceptability by a significant portion of users

(8) Improved ability to use FORTRAN programs and data in conjunction with other languages and environments

A2 Conflicts with ANSI X3.9-1966

An extremely important consideration in the preparation of this standard was the minimization of conflicts with the previous standard, ANSI X3.9-1966. This standard includes changes that create conflicts with ANSI X3.9-1966 only when such changes were necessary to correct an error in the previous standard or to add to the power of the FORTRAN language in a significant manner. The following is a list of known conflicts:

(1) A line that contains only blank characters in columns 1 through 72 is a comment line. ANSI X3.9-1966 allowed such a line to be the initial line of a statement.

(2) Columns 1 through 5 of a continuation line must contain blanks. A published interpretation of ANSI X3.9-1966 specified that columns 1-5 of a continuation line may contain any character from the FORTRAN character set except that column 1 must not contain a C.

(3) Hollerith constants and Hollerith data are not permitted in this standard. ANSI X3.9-1966 permitted the use of Hollerith constants in DATA and CALL

statements, the use of noncharacter list items in formatted input/output statements with A edit descriptors, and the referencing of noncharacter arrays as formats. Note that the H edit (field) descriptor is permitted; it is not a Hollerith constant.

(4) The value of each comma-separated subscript expression in a subscript must not exceed its corresponding upper bound declared for the array name in the program unit. In the example:

```
DIMENSION A(10,5)
Y=A(11,1)
```

The reference to A(11,1) is not permitted for the array A(10,5). ANSI X3.9-1966 permitted a subscript expression to exceed its corresponding upper bound if the maximum subscript value for the array was not exceeded.

(5) Only an array that is declared as a one-dimensional array in the program unit may have a one-dimensional subscript in an EQUIVALENCE statement. In the example:

```
DIMENSION B(2,3,4), C(4,8)
EQUIVALENCE (B(23), C(1,1))
```

B(23) is not permitted. ANSI X3.9-1966 permitted arrays that were declared as two- or three-dimensional arrays to appear in an EQUIVALENCE statement with a one-dimensional subscript.

(6) A name must not have its type explicitly specified more than once in a program unit. ANSI X3.9-1966 did not explicitly have such a prohibition.

(7) This standard does not permit a transfer of control into the range of a DO-loop from outside the range. The range of a DO-loop may be entered only by the execution of a DO statement. ANSI X3.9-1966 permitted transfer of control into the range of a DO-loop under certain conditions. This involved the concept referred to as "extended range of a DO."

(8) A labeled END statement could conflict with the initial line of a statement in an ANSI X3.9-1966 standard-conforming program.

(9) A record must not be written after an endfile record in a sequential file. ANSI X3.9-1966 did not prohibit this, but provided no interpretation for the reading of an endfile record.

(10) A sequential file may not contain both formatted and unformatted records. A published interpretation of ANSI X3.9-1966 specified that this was permitted.

(11) Negative values for input/output unit identifiers are prohibited in this standard. ANSI X3.9-1966 did not explicitly prohibit them for variable unit identifiers.

(12) A simple I/O list enclosed in parentheses is prohibited from appearing in an I/O list.

This requires that parentheses enclosing more than one I/O list item must mark an implied DO-loop. The restriction was imposed to eliminate potential syntactic ambiguities introduced by complex constants in list-directed output lists. As all the parentheses referred to are redundant, a program can be made conforming with this standard by deleting redundant parentheses enclosing more than one list item in an I/O list.

(13) The definition of an entity associated with an entity in an input list occurs at the same time as the definition of the list entity. ANSI X3.9-1966 delayed the definition of such an associated entity until the end of execution of the input statement.

(14) Reading into an H edit (field) descriptor in a FORMAT statement is prohibited in this standard.

(15) The range of a scale factor for E, D, and G output fields is restricted to reasonable values. ANSI X3.9-1966 had no such restriction, but did not provide a clear interpretation of the meaning of the unreasonable values.

(16) A processor must not produce a numeric output field containing a negative zero. ANSI X3.9-1966 required this if the internal value of a real or double precision datum was negative.

(17) On output, the I edit descriptor must not produce unnecessary leading zeros.

(18) On output, the F edit descriptor must not produce unnecessary leading zeros, other than the optional leading zero for a value less than one.

(19) Following the E or D in an E or D output field, a + or - is required immediately prior to the exponent field. This improves compatibility with American National Standard for the Representation of Numeric Values in Character Strings for Information Interchange, ANSI X3.42-1975. ANSI X3.9-1966

permitted a blank as a replacement for + in the exponent sign.

(20) An intrinsic function name that is used as an actual argument must appear in an INTRINSIC statement rather than an EXTERNAL statement. Note that the intrinsic function class includes the basic external function class of ANSI X3.9-1966.

(21) The appearance of an intrinsic function name in a type-statement that conflicts with the type specified in Table 5 is not sufficient to remove the name from the intrinsic function class. In ANSI X3.9-1966, this condition was sufficient to remove the name from the intrinsic function class.

(22) More intrinsic function names have been added and could conflict with the names of subprograms. These names are ACOS, ANINT, ASIN, CHAR, COSH, DACOS, DASIN, DCOSH, DDIM, DINT, DNINT, DPROD, DSINH, DTAN, DTANH, ICHAR, IDNINT, INDEX, LEN, LGE, LGT, LLE, LLT, LOG, LOG10, MAX, MIN, NINT, SINH, and TAN.

(23) The units of the arguments and results of the intrinsic functions (and basic external functions) were not specified in ANSI X3.9-1966 and are specified in this standard. The range of the arguments and results has also been specified. These specifications may be different from those used on some processors conforming to ANSI X3.9-1966.

(24) An executable program must not contain more than one unnamed block data subprogram. ANSI X3.9-1966 did not have this prohibition and could be interpreted to permit more than one.

A3 Standard Items That Inhibit Portability

Although the primary purpose of this standard is to promote portability of FORTRAN programs, there are some items in it that tend to inhibit portability.

(1) Procedures written in languages other than FORTRAN may not be portable.

(2) Because the collating sequence has not been completely specified, character relational expressions do not necessarily have the same value on all processors. However, the intrinsic functions LGE, LGT, LLE, and LLT can be used to provide a more portable comparison of character entities.

(3) Character data, H edit descriptors, apostrophe edit descriptors, and comment lines may include characters that are acceptable to one processor but unacceptable to another processor.

(4) No explicit requirements are specified for file names. A file name that is acceptable to one processor may be unacceptable to another processor.

(5) Input/output unit numbers and unit capabilities may vary among processors.

A4 Recommendation for Enhancing Portability

To enhance the development of portable FORTRAN programs, a producer should provide some means of identifying nonstandard syntax supported by his processor. Alternatives for doing this include appropriate documentation, features of the processor, and other means.

APPENDIX B: SECTION NOTES

B1 Section 1 Notes

What this standard calls a "processor" is any mechanism that can carry out the actions of a program. Commonly, this may be any of these:

(1) The combined actions of a computer (hardware), its operating system, a compiler, and a loader

(2) An interpreter

(3) The mind of a human, perhaps with the help of paper and pencil

When you read this standard, it is important to keep its point of view in mind. The standard is written from the point of view of a programmer using the language, and not from the point of view of the implementation of a processor. This point of view affects the way you should interpret the standard. For example, in 3.3 the assertion is made:

> "... a statement must contain no more than 1320 characters."

This means that if a programmer writes a longer statement, his program is not standard conforming. Therefore, it will get different treatment on different processors. Some processors will accept the program, and some will not. Some may even seemingly accept the program but process it incorrectly. The assertion means that all standard-conforming processors must accept statements up to 1320 characters long. That is the only inference about a standard-conforming processor that can be made from the assertion.

The assertion does not mean that a standard-conforming processor is prohibited from accepting longer statements. Accepting longer statements would be an extension.

The assertion does not mean that a standard-conforming processor must diagnose statements longer than 1320 characters, although it may do so.

In general, a standard-conforming processor is one that accepts all standard-conforming programs and processes them according to the rules of this standard. Thus, the specification of a standard-conforming processor must be inferred from this document.

In some places, explicit prohibitions or restrictions are stated, such as the above statement-length restriction. Such assertions restrict what programmers can write in standard-conforming programs and have no more weight in the standard than an omitted feature. For example, there is no mention anywhere in the standard of double precision integers. Because it is omitted, programmers must not use this feature in standard-conforming programs. A standard-conforming processor may or may not provide it or diagnose its use. Thus, an explicit prohibition (such as statements longer than 1320 characters) and an omission (such as double precision integers) are equivalent in this standard.

B2 Section 2 Notes

Some of the terminology used in this document is different from that used to describe other programming languages. The following indicates terms from other languages that are approximately equvialent to some FORTRAN terms.

FORTRAN	Other Languages
Variable	Simple Variable
Array Element	Subscripted Variable
Subscript Expression	Subscript
Subscript	(none)
Dummy Argument	Formal Argument, Formal Parameter
Actual Argument	Actual Parameter

In particular, the FORTRAN terms "subscript" and "subscript expression" should be studied carefully by readers who are unfamiliar with this standard (5.4).

The term "symbolic name" is frequently shortened to "name" throughout the standard.

B3 Section 3 Notes

A partial collating sequence is specified. If possible, a processor should use the American National Standard Code for Information Interchange, ANSI X3.4-1977 (ASCII), sequence for the complete FORTRAN character set.

When a continuation line follows a comment line, the continuation line is part of the current statement; it is not a continuation of the comment line. A comment line is not part of a statement.

The standard does not restrict the number of consecutive comment lines. The limit of 19 continuation lines permitted for a statement should not be construed as being a limitation on the number of consecutive comment lines.

There are 99999 unique statement labels and a processor must accept 99999 as a statement label. However, a processor may have an implementation limit on the total number of unique statement labels in one program unit (3.4).

Blanks and leading zeros are not significant in distinguishing between statement labels. For example, 123, 1 23, and 0123 are all forms of the same statement label.

B4 Section 4 Notes

A processor must not consider a negative zero to be different from a positive zero.

ANSI X3.9-1966 used the term "constant" to mean an unsigned constant. This standard uses the term "constant" to have its more normal meaning of an optionally signed constant when describing arithmetic constants. The term "unsigned constant" is used wherever a leading sign is not permitted on an arithmetic constant.

A character constant is a representation of a character value. The delimiting apostrophes are part of the representation but not part of the value; double apostrophes are used to represent a single embedded apostrophe. For example:

Character Constant	Character Value
'CAT'	CAT
'ISN''T'	ISN'T
'''ISN''''T'''	'ISN''T'

Note that the value of the character constant '''ISN''''T''' is a representation of another character constant.

Some programs that used an extension to ANSI X3.9-1966 that permitted a Hollerith constant delimited by apostrophes instead of the nH form do not conform to this standard.

B5 Section 5 Notes

For the array declarator A(2,3), the use of the array name A in the proper context, such as in an input/output list, specifies the following order for the array elements: A(1,1), A(2,1), A(1,2), A(2,2), A(1,3), A(2,3).

B6 Section 6 Notes

If V is a variable name, the interpretation and value of V, +V, and (V) are the same. However, the three forms may not always be used interchangeably. For example, the forms +V and (V) may not be used as list items of a READ statement or as actual arguments of a procedure reference if the procedure defines the corresponding dummy argument.

B7 Section 7 Notes

Although DIMENSION statements, type-statements, and statement function statements are classified as nonexecutable statements, they may contain references that are executed. Expressions containing variables in DIMENSION statements and type-statements may be evaluated whenever a reference to the program unit is executed. The expression in a statement function statement is evaluated whenever a function reference to the statement function is executed.

B8 Section 8 Notes

If a processor allows a one-dimensional subscript for a multidimensional array in an EQUIVALENCE statement, the interpretation should be as though the subscript expression were the leftmost one and the missing subscript expressions each have their respective lower dimension bound value.

ANSI X3.9-1966 permitted two- and three-dimensional arrays to have a one-dimensional subscript in an EQUIVALENCE statement. The following table can be used to convert a one-dimensional subscript to the corresponding multidimensional subscript:

n	Dimension	Subscript Value	Subscript
1	(d_1)	s	(s)
2	(d_1,d_2)	s	$(1+MOD(s-1,d_1), 1+(s-1)/d_1)$
3	(d_1,d_2,d_3)	s	$(1+MOD(s-1,d_1), 1+MOD((s-1)/d_1,d_2), 1+(s-1)/(d_1*d_2))$

Each expression in the last column of the table is evaluated according to the rules for integer expressions.

A processor that allows additional intrinsic functions should allow their names to appear in an INTRINSIC statement.

As an extension to ANSI X3.9-1966, many processors permitted the retention of certain values at the completion of execution of a subprogram, such as local variables and arrays, initially defined data that had been changed, and named common blocks not specified in the main program, whereas other processors prohibited the retention of such values. In ANSI X3.9-1966 such entities were undefined at the completion of execution of the subprogram, and therefore a standard-conforming program could not retain these values. The SAVE statement provides a facility for data retention.

B9 Section 9 Notes

An entity is "initially defined" only by a DATA statement. An assignment statement may define or redefine an entity but it does not "initially define" the entity.

Initially defined entities in a subprogram may become undefined at the execution of a RETURN or END statement if they are assigned any value, including their initial value, during the execution of the executable program (see 8.9 and 15.8.4).

B10 Section 10 Notes

All four types of implied arithmetic conversion are permitted in an arithmetic assignment statement.

B11 Section 11 Notes

A logical IF statement must not contain another logical IF statement or a block IF statement; however, it may contain an arithmetic IF statement. The following is allowed:

IF (logical expr.) IF (arithmetic expr.) $\underline{s}_1,\underline{s}_2,\underline{s}_3$

A processor is not required to evaluate the iteration count in a DO-loop if the same effect is achieved without evaluation. However, the processor must allow redefinition of variables and array elements that appear after the equals in a DO statement during the execution of the DO-loop without affecting the number of times the DO-loop is executed and without affecting the value by which the DO-variable is incremented.

If J1 > J2, ANSI X3.9-1966 does not allow execution of the following DO statement:

```
DO 100 J=J1,J2
```

Some processors that allowed such a case executed the range of the DO-loop once, whereas other processors did not execute the range of the DO-loop. This standard allows such a case and requires that the processor execute the range of the DO-loop zero times. The following change to the DO statement will require that the processor execute the range at least once:

```
DO 100 J=J1,MAX(J1,J2)
```

References to function procedures and subroutine procedures may appear within the range of a DO-loop or within an IF-block, ELSE IF-block, or ELSE-block. Execution of a function reference or a CALL statement is not considered a transfer of control in the program unit that contains the reference, except when control is returned to a statement identified by an alternate return specifier in a CALL statement. Execution of a RETURN or END statement in a referenced procedure, or execution of a transfer of control within a referenced procedure, is not considered a transfer of control in the program unit that contains the reference.

The CONTINUE statement is an executable statement that has no effect of itself. It can serve as an executable statement on which to place a statement label when no effect of execution is desired. For example, it can serve as the statement referred to by a GO TO statement or as the terminal statement of a DO-loop. Although the CONTINUE statement has no effect of itself, it causes execution to continue with incrementation processing when it is the terminal statement of a DO-loop.

The standard does not define the term "accessible" in the STOP or PAUSE statement in order to allow a wide latitude in adapting to a processor environment. Some processors may use the _n_ in the PAUSE or STOP statement for documentation only. Other processors may display the _n_ to the user or to the operator. In order not to confine its use, the meaning of "accessible" is purposely left vague.

B12 Section 12 Notes

What is called a "record" in FORTRAN is commonly called a "logical record." There is no concept in FORTRAN of a "physical record."

An endfile record does not necessarily have any physical embodiment. The processor may use a record count or other means to register the position of the file at the time an ENDFILE statement is executed, so that it can take appropriate action when that position is again reached during a read operation. The endfile record, however it is implemented, is considered to exist for the BACKSPACE statement.

An internal file permits data to be transferred with conversion between internal storage areas using the READ and WRITE statements. This facility was implemented as an extension to ANSI X3.9-1966 on many processors as ENCODE and DECODE statements. Specifying the READ and WRITE statements to perform this process avoids such confusion as: "Is ENCODE like READ or is it like WRITE?"

This standard accommodates, but it does not require, file cataloging. To do this, several concepts are introduced.

In ANSI X3.9-1966 many properties were given to a unit that in this standard are given to the connection of a file to a unit. Also, additional properties are introduced.

Before any input/output can be performed on a file, it must be connected to a unit. The unit then serves as a designator for that file as long as it is connected. To be connected does not imply that "buffers" have or have not been allocated, that "file-control tables" have or have not been filled out, or that any other method of implementation has been used. Connection means that (barring some other fault) a READ or WRITE statement can be executed on the unit, hence on the file. Without a connection, a READ or WRITE statement cannot be executed.

Totally independent of the connection state is the property of existence, this being a file property. The processor "knows" of a set of files that exist at a given time for a given executable program. This set would include tapes ready to read, files in a catalog, a keyboard, a printer, etc. The set may exclude files inaccessible to the executable program because of security, because they are already in use by another executable program, etc. This standard does not specify which files exist, hence wide latitude is available to a processer to implement security, locks, privilege techniques, etc. Existence is a convenient concept to designate all of the files that an executable program can potentially process.

All four combinations of connection and existence may occur:

Connect	Exist	Examples
Yes	Yes	A card reader loaded and ready to be read
Yes	No	A printer before the first line is written
No	Yes	A file named 'JOE' in the catalog
No	No	A reel of tape destroyed in the fire last week

Means are provided to create, delete, connect, and disconnect files.

A file may have a name. The form of a file name is not specified. If a system does not have some form of cataloging or tape labeling for at least some of its files, all file names will disappear at the termination of execution. This is a valid implementation. Nowhere does this standard require names to survive for any period of time longer than the execution time span of an executable program. Therefore, this standard does not impose cataloging as a prerequisite. The naming feature is intended to allow use of a cataloging system where one exists.

A file may become connected to a unit in either of two ways: preconnection or execution of an OPEN statement. Preconnection is performed prior to the beginning of execution of an executable program by means external to FORTRAN. For example, it may be done by job control action or by processor established defaults. Execution of an OPEN statement is not required to access preconnected files.

The OPEN statement provides a means to access existing files that are not preconnected. An OPEN statement may be used in either of two ways: with a file name (open by name) and without a file name (open by unit). A unit is given in either case. Open by name connects the specified file to the specified unit. Open by unit connects a processor-determined default file to the specified unit. (The default file may or may not have a name.)

Therefore, there are three ways a file may become connected and hence processed: preconnection, open by name, and open by unit. Once a file is connected, there is no means in standard FORTRAN to determine how it became connected.

In subset FORTRAN, sequential access may be performed only on preconnected files, and direct access only on files that are opened by unit.

An OPEN statement may also be used to create a new file. In fact, any of the foregoing three connection methods may be performed on a file that does not exist. When a unit is preconnected, writing the first record creates the file. With the other two methods, execution of the OPEN statement creates the file.

When a unit becomes connected to a file, either by execution of an OPEN statement or by preconnection, the following connection properties may be established:

(1) An access method, which is sequential or direct, is established for the connection.

(2) A form, which is formatted or unformatted, is established for a connection to a file that exists or is created by the connection. For a connection that results from execution of an OPEN statement, a default form (which depends on the access method, as described in 12.10.1) is established if no form is specified. For a preconnected file that exists, a form is established by preconnection. For a preconnected file that does not exist, a form may be established, or the establishment of a form may be delayed until the file is created (for example, by execution of a formatted or unformatted WRITE statement).

(3) A record length may be established. If the access method is direct, the connection establishes a record

length, which specifies the length of each record of the file. A connection for sequential access does not have this property.

(4) A blank significance property, which is ZERO or NULL, is established for a connection for which the form is formatted. This property has no effect on output. For a connection that results from execution of an OPEN statement, the blank significance property is NULL by default if no blank significance property is specified. For a preconnected file, the property is established by preconnection.

The blank significance property of the connection is effective at the beginning of each formatted input statement. During execution of the statement, any BN or BZ edit descriptors encountered may temporarily change the effect of embedded and trailing blanks.

A processor has wide latitude in adapting these concepts and actions to its own cataloging and job control conventions. Some processors may require job control action to specify the set of files that exist or that will be created by an executable program. Some processors may require no job control action prior to execution. This standard enables processors to perform a dynamic open, close, and file creation, but it does not require such capabilities of the processor.

The meaning of "open" in contexts other than FORTRAN may include such things as mounting a tape, console messages, spooling, label checking, security checking, etc. These actions may occur upon job control action external to FORTRAN, upon execution of an OPEN statement, or upon execution of the first read or write of the file. The OPEN statement describes properties of the connection to the file and may or may not cause physical activities to take place. It is a place for an implementation to define properties of a file beyond those required in standard FORTRAN.

Similarly, the actions of dismounting a tape, protection, etc. of a "close" may be implicit at the end of a run. The CLOSE statement may or may not cause such actions to occur. This is another place to extend file properties beyond those of standard FORTRAN. Note, however, that the execution of a CLOSE statement on unit 10 followed by an OPEN statement on the same unit to the same file or to a different file is a permissible sequence of events. The processor may not deny this sequence solely because the implementation chooses to do the physical act of closing the file at the termination of execution of the program.

This standard does not address problems of security, protection, locking, and many other concepts that may be part of the concept of "right of access." Such concepts are considered to be in the province of an operating system.

The OPEN and INQUIRE statements can be extended naturally to consider these things.

Possible access methods for a file are: sequential and direct. The processor may implement two different types of files, each with its own access method. It may also implement one type of file with two different access methods.

Direct access to files is of a simple and commonly available type, that is, fixed-length records. The key is a positive integer.

Keyword forms of specifiers are used because there are many specifiers and a positional notation is difficult to remember. The keyword form sets a style for processor extensions. The UNIT= and FMT= keywords are offered for completeness, but their use is optional. Thus, compatibility with ANSI X3.9-1966 is achieved.

Format specifications may be included in READ and WRITE statements, as in:

```
READ ( UNIT=10, FMT='(I3,A4,F10.2)' ) K,ALPH,X
```

ANSI X3.9-1966 allowed a standard-conforming program to write an endfile record but did not allow the reading of an endfile record. In this standard, the END= specifier allows end-of-file detection and continuation of execution of the program.

List-directed input/output allows data editing according to the type of the list item instead of by a format specifier. It also allows data to be free-field, that is, separated by commas or blanks.

List-directed input/output is record oriented to or from a formatted sequential file. Each read or write begins with a new record. The form of list-directed data on a sequential output file is not necessarily suitable for list-directed input. However, there are no mandatory errors specified for reading list-directed data previously written. The results may not be guaranteed because of the syntax using apostrophes for character data or the r*c form of a repeated constant. All other applications should work, and attempting to read previously written list-directed output is not prohibited in a standard-conforming program.

If no list items are specified in a list-directed input/output statement, one input record is skipped or one empty output record is written.

An example of a restriction on input/output statements (12.12) is that an input statement may not specify that data are to be read from a printer.

B13 Section 13 Notes

The term "edit descriptor" in this standard was "field descriptor" in ANSI X3.9-1966.

If a character constant is used as a format identifier in an input/output statement, care must be taken that the value of the character constant is a valid format specification. In particular, if the format specification contains an apostrophe edit descriptor, two apostrophes must be written to delimit the apostrophe edit descriptor and four apostrophes must be written for each apostrophe that occurs within the apostrophe edit descriptor. For example, the text:

```
          2 ISN'T 3
```

may be written by various combinations of output statements and format specifications:

```
          WRITE(6,100) 2,3
      100 FORMAT(1X,I1,1X,'ISN''T',1X,I1)

          WRITE(6,'(1X,I1,1X,''ISN''''T'',1X,I1)' ) 2,3

          WRITE(6,200) 2,3
      200 FORMAT(1X,I1,1X,5HISN'T,1X,I1)

          WRITE(6,'(1X,I1,1X,5HISN''T,1X,I1)' ) 2,3

          WRITE(6,'(A)') ' 2 ISN''T 3'

          WRITE(6,'(1X,I1,A,I1)') 2, ' ISN''T ', 3
```

Note that two consecutive apostrophes in an H edit descriptor within a character constant are counted as only one Hollerith character.

The T edit descriptor includes the carriage control character in lines that are to be printed. T1 specifies the carriage control character, and T2 specifies the first character that is printed.

The length of a record is not always specified exactly and may be processor dependent.

The number of records read by a formatted input statement can be determined from the following rule: A record is read at the beginning of the format scan (even if the input list is empty), at each slash edit descriptor encountered in the format, and when a format rescan occurs at the end of the format.

The number of records written by a formatted output statement can be determined from the following rule: A record is written when a slash edit descriptor is encountered in the format, when a format rescan occurs at the end of the format, and at completion of execution of the output statement (even if the output list is empty). Thus, the occurrence of _n_ successive slashes between two other edit descriptors causes _n_ - 1 blank lines if the records are printed. The occurrence of _n_ slashes at the beginning or end of a complete format specification causes _n_ blank lines if the records are printed. However, a complete format specification containing _n_ slashes (_n_ ≥ 0) and no other edit descriptors causes _n_ + 1 blank lines if the records are printed. For example, the statements

```
      PRINT 3
    3 FORMAT(/)
```

will write two records that cause two blank lines if the records are printed.

The following examples illustrate list-directed input. A blank character is represented by _b_.

Example 1:

```
Program:    J=3
            READ *,I
            READ *,J
```

Sequential input file:

record 1: _b_1_b_,4_bbbbb_
record 2: ,2_bbbbbbbb_

Result: I=1, J=3

Explanation: The second READ statement reads the second record. The initial comma in the record designates a null value; therefore, J is not redefined.

Example 2:

```
Program:    CHARACTER A*8, B*1
            READ *, A, B
```

Sequential input file:

record 1: '_bbbbbbbb_'
record 2: 'QXY'_b_'Z'

Result: A='_bbbbbbbb_', B='Q'

Explanation: The end of a record cannot occur between two apostrophes representing an embedded apostrophe in a character constant; therefore, A is set to the character constant '_bbbbbbbb_'. The end of a record acts as a blank, which in this case is a value separator because it occurs between two constants.

B14 Section 14 Notes

The name of a main program has no explicit use within the FORTRAN language. It is available for documentation and for possible use within a computer environment.

B15 Section 15 Notes

A FUNCTION statement specifies the name of an external function, and each ENTRY statement in a function subprogram specifies an additional external function name. A SUBROUTINE statement specifies the name of a subroutine, and each ENTRY statement in a subroutine subprogram specifies an additional subroutine name.

The intrinsic function names IFIX, IDINT, FLOAT, and SNGL have been retained to support programs that conform to ANSI X3.9-1966. However, future use of these intrinsic function names is not recommended.

For the specific functions that define the maximum and minimum values with a function type different from the argument type (AMAX0, MAX1, AMIN0, and MIN1), it is recommended that an expression containing the generic name preceded by a type conversion function be used, for example, REAL(MAX($\underline{a}_1$, $\underline{a}_2$, ...)) for AMAX0($\underline{a}_1$, $\underline{a}_2$, ...), so that these specific function names may be deleted in a future revision of this standard.

This standard provides that a standard-conforming processor may supply intrinsic functions in addition to those defined in Table 5 (15.10). Because of this, care must be taken when a program is used on more than one processor because a function name not in Table 5 may be classified as an external function name on one processor and as an intrinsic function name on another processor in the absence of a declaration for that name in an EXTERNAL or INTRINSIC statement.

To guard against this possibility, it is suggested that any external functions referenced in a program should appear in an EXTERNAL statement in every program unit in which a reference to that function appears. If a program unit references a processor-supplied intrinsic function that does not appear in Table 5, the name of the function should appear in an INTRINSIC statement in the program unit.

The distinction between external functions (user defined) and intrinsic functions (processor defined) may be clarified by the following table:

	Different Processor Definitions (Table 5 extended)		
	Processor 1	Processor 2	Processor 3
Different User Specifications	Intrinsic Integer FROG	Intrinsic Complex FROG	(none)
Y=FROG(A)	Intrinsic Integer FROG	Intrinsic Complex FROG	External Real FROG
INTRINSIC FROG Y=FROG(A)	Intrinsic Integer FROG	Intrinsic Complex FROG	Undefined
INTEGER FROG Y=FROG(A)	Intrinsic Integer FROG	Undefined	External Integer FROG
INTRINSIC FROG INTEGER FROG Y=FROG(A)	Intrinsic Integer FROG	Undefined	Undefined
EXTERNAL FROG Y=FROG(A)	External Real FROG	External Real FROG	External Real FROG
EXTERNAL FROG INTEGER FROG Y=FROG(A)	External Integer FROG	External Integer FROG	External Integer FROG

If a generic name is the same as the specific name of an intrinsic function for a specified type of argument, a reference to the function with an argument of that type may be considered to be either a specific or generic function reference.

The use of the concatenation operator with operands of nonconstant length has been restricted to the assignment statement so that a processor need not implement dynamic storage allocation.

When a character array is an actual argument, the array is considered to be one string of characters and there need not be correspondence between the actual array elements and the dummy array elements. Only subset FORTRAN requires such correspondence.

The intrinsic functions ICHAR and CHAR provide a means of converting between a character and an integer, based on the position of the character in the processor collating sequence. The first character in the collating sequence corresponds to position 0 and the last to position _n_ - 1, where _n_ is the number of characters in the collating sequence.

Many processors provide a collating sequence that is the same as the ordering of the internal representation of the character (where the internal representation may be regarded as either a representation of a character or of some integer). For example, for a seven-bit character, the internal representation of the first character is '0000000' binary (0 decimal) and the last character is '1111111' binary (127 decimal). For such a processor, ICHAR returns the value of an internal character representation, considered as an integer. CHAR takes an appropriate small integer and returns the character having the same internal representation.

B16 Section 16 Notes

The name of a block data subprogram has no explicit use within the FORTRAN language. It is available for documentation and for possible use within a computer environment.

B17 Section 17 Notes

The size of an array is the number of elements (5.2.3), but the storage sequence of the array also has a size, which may be different from the number of elements (17.1.1).

The definition of character entities occurs on a character-by-character basis. The use of substrings or partially associated entities permits individual characters or groups of characters within an entity to become defined or undefined.

B18 Section 18 Notes

There is no explicit means for declaring an entity to be a variable. An entity becomes a variable if it is used in a manner that does not cause it to be exclusively something else. Note that the name of a variable may also be the name of a common block, except when the name of the variable is also the name of a function.

APPENDIX C: HOLLERITH

The character data type was added to provide a character data processing capability that is superior to the Hollerith data capability that existed in ANSI X3.9-1966.

The Hollerith data type has been deleted. For processors that extend the standard by allowing Hollerith data, the following rules for programs are recommended:

C1 Hollerith Data Type

Hollerith is a data type; however, a symbolic name must not be of type Hollerith. Hollerith data, other than constants, are identified under the guise of a name of type integer, real, or logical. They must not be identified under the guise of type character. No recommendation is made regarding Hollerith under the guise of double precision or complex.

A Hollerith datum is a string of characters. The string may consist of any characters capable of representation in the processor. The blank character is significant in a Hollerith datum. Hollerith data may have an internal representation that is different from that of other data types.

An entity of type integer, real, or logical may be defined with a Hollerith value by means of a DATA statement (C4) or READ statement (C6). When an entity is defined with a Hollerith value, its totally associated entities are also defined with that Hollerith value. When an entity of type integer, real, or logical is defined with a Hollerith value, the entity and its associates become undefined for use as an integer, real, or logical datum.

C2 Hollerith Constant

The form of a Hollerith constant is a nonzero, unsigned, integer constant _n_ followed by the letter H, followed by a string of exactly _n_ contiguous characters. The string may consist of any characters capable of representation in the processor. The string of _n_ characters is the Hollerith datum.

In a Hollerith constant, blanks are significant only in the _n_ characters following the letter H.

C3 Restrictions on Hollerith Constants

A Hollerith constant may appear only in a DATA statement and in the argument list of a CALL statement.

C4 Hollerith Constant in a DATA Statement

An integer, real, or logical entity may be initially defined with a Hollerith datum by a DATA statement.

A Hollerith constant may appear in the list *clist*, and the corresponding entity in the list *nlist* may be of type integer, real, or logical.

For an entity of type integer, real, or logical, the number of characters *n* in the corresponding Hollerith constant must be less than or equal to *g*, where *g* is the maximum number of characters that can be stored in a single numeric storage unit at one time. If *n* is less than *g*, the entity is initially defined with the *n* Hollerith characters extended on the right with *g* - *n* blank characters.

Note that each Hollerith constant initially defines exactly one variable or array element. Also note that *g* is processor dependent.

C5 Hollerith Format Specification

A format specification may be an array name of type integer, real, or logical.

The leftmost characters of the specified entity must contain Hollerith data that constitute a format specification when the statement is executed.

The format specification must be of the form described in 13.2. It must begin with a left parenthesis and must end with a right parenthesis. Data may follow the right parenthesis that ends the format specification and have no effect. Blank characters may precede the format specification.

A Hollerith format specification must not contain an apostrophe edit descriptor or an H edit descriptor.

C6 A Editing of Hollerith Data

The A*w* edit descriptor may be used with Hollerith data when the input/output list item is of type integer, real, or logical. On input, the input list item will become defined with Hollerith data. On output, the list item must be defined with Hollerith data.

Editing is as described for A*w* editing of character data except that *len* is the maximum number of characters that can be stored in a single numeric storage unit.

C7 Hollerith Constant in a Subroutine Reference

An actual argument in a subroutine reference may be a Hollerith constant. The corresponding dummy argument must be of type integer, real, or logical. Note that this is an exception to the rule that requires that the type of the actual and dummy argument must agree.

APPENDIX D: SUBSET OVERVIEW

This Appendix provides an overview of the two levels of FORTRAN specified in this standard, including the general criteria used for including or excluding a feature at a given level, and a section-by-section summary of the principal differences between the full language and the subset.

D1 Background

The full FORTRAN language described in this document is a superset of the FORTRAN language described in ANSI X3.9-1966, with the exceptions previously noted. In formulating a subset philosophy, the following existing FORTRAN standards were considered:

(1) American National Standard FORTRAN, ANSI X3.9-1966

(2) American National Standard Basic FORTRAN, ANSI X3.10-1966

(3) International Standard Programming Language FORTRAN, ISO R1539

The ISO R1539 document describes three levels: basic, intermediate, and full. The ISO R1539 basic level corresponds closely with ANSI X3.10-1966; the ISO R1539 full level corresponds closely with ANSI X3.9-1966; and the ISO R1539 intermediate level is in between.

It was thought that the ISO R1539 basic level and the ANSI X3.10-1966 had not been sufficiently used, even on small computer systems, to warrant a subset corresponding to that level.

The ISO R1539 intermediate level has been sufficiently used to warrant a subset of similar capability.

However, it was also thought that some of the capabilities in the full language described here, but not part of any current standard or recommendation, are so important for the general use of the language that they should be present in the subset, at least to some degree.

Furthermore, it was thought that the specification of ANSI X3.10-1966 in such a manner that it is not a subset of ANSI X3.9-1966 was inconsistent with the primary goal of promoting program interchange. Consequently, careful attention has been given to ensuring that a program that conforms to the subset of this standard will also conform to the full language.

D2 Criteria

The criteria in D2.1 and D2.2 were adopted for the two levels of FORTRAN within this standard.

D2.1 Full Language. The most notable new elements of the full language that have been included at both levels are: character data type, mixed-type arithmetic, INTRINSIC statement, SAVE statement, and direct access I/O statements.

D2.2 Subset Language

(1) The subset must be a proper subset of the full language.

(2) The subset must be based on ISO R1539 intermediate level FORTRAN.

(3) The subset must include, at a fundamental level, those features of the full language that significantly increase the scope of the language.

(4) The elements of the subset must make a minimum demand on storage requirements, particularly during execution.

(5) The subset must require a minimum of effort for the development and maintenance of a viable FORTRAN processor.

D3 Summary of Subset Differences

This section summarizes the differences between the full language and the subset in this standard. It is organized primarily on the basis of the standard itself. The differences are discussed under the section where each language element is primarily presented. Of course, a difference in one section may cause changes in other sections. Such changes are not noted here.

An exception to the above practice is the subsetting of the character data type. The description of character data type and its usage is so distributed throughout the standard that a more meaningful summary is produced by collecting the relevant items into a single presentation.

D3.1 Section 1: Introduction. The subset is the same as the full language (see also D4).

D3.2 Section 2: FORTRAN Terms and Concepts. The subset is the same as the full language.

D3.3 Section 3: Characters, Lines, and Execution Sequence. The subset is the same as the full language except that:

(1) The character set does not include the currency symbol ($) or the colon (:).

(2) Statements may have up to nine continuation lines.

(3) DATA statements must follow all specification statements and precede all statement function statements and executable statements.

(4) A comment line must not precede a continuation line.

D3.4 Section 4: Data Types and Constants. The subset is the same as the full language except that double precision and complex data types are not included. Note that each entity of type character must have a constant length.

D3.5 Section 5: Arrays and Substrings. The subset is the same as the full language except that:

(1) An array declarator must not have an explicit lower bound.

(2) A dimension declarator must be either an integer constant or an integer variable. (This excludes integer expressions, but allows a variable in common.)

(3) An array may have up to three dimensions.

(4) A subscript expression may be an expression containing only integer variables and constants. (This excludes function and array element references.)

D3.6 Section 6: Expressions. The subset is the same as the full language except that a constant expression is allowed only where a general expression is allowed, the logical operators .EQV. and .NEQV. are not included, and there are restrictions on character expressions as described in D3.19.

D3.7 Section 7: Executable and Nonexecutable Statement Classification. The classification of a statement in the subset is the same as in the full language. However, the subset does not include PRINT, CLOSE, INQUIRE, ENTRY, BLOCK DATA, PARAMETER, DOUBLE PRECISION, and COMPLEX statements.

D3.8 Section 8: Specification Statements. The subset is the same as the full language except that:

(1) The PARAMETER statement is not included.

(2) Only the names of common blocks (enclosed in slashes) may appear in the list of a SAVE statement. The form of the SAVE statement without a list is not included.

D3.9 <u>Section 9: DATA Statement</u>. The subset is the same as the full language except that:

(1) Only names of variables, arrays, and array elements are allowed in the list <u>nlist</u>. Implied-DO lists are not included.

(2) Values in the list <u>clist</u> must agree in type with the corresponding item in the list <u>nlist</u>. Type conversion is not included.

Note that DATA statements must follow all specification statements and precede all statement function statements and executable statements.

D3.10 <u>Section 10: Assignment Statements</u>. The subset is the same as the full language except for restrictions on character type presented in D3.19.

D3.11 <u>Section 11: Control Statements</u>. The subset is the same as the full language except that:

(1) A DO-variable must be an integer variable and DO parameters must be integer constants or integer variables.

(2) In a computed GOTO statement, the index expression must be an integer variable.

D3.12 <u>Section 12: Input/Output Statements</u>. The subset is the same as the full language except that:

(1) The CLOSE statement is not included.

(2) The INQUIRE statement is not included.

(3) List-directed READ and WRITE statements are not included.

(4) An internal file identifier must be a character variable or character array element.

(5) Formatted direct access files and statements are not included.

(6) External unit identifiers must be an integer constant or integer variable.

(7) A format identifier must be the label of a FORMAT statement, an integer variable that has been assigned the label of a FORMAT statement, or a character constant.

(8) The UNIT= and FMT= forms of unit and format specifiers are not included.

(9) The ERR= specifier is not included.

(10) The forms READ f [,iolist] and PRINT f [,iolist] are not included.

(11) In input/output lists, the implied-DO parameters must be integer constants and variables. Implied-DO-variables must be of type integer.

(12) Variable names, array element names, and array names may appear as input/output list items; constants, character substring references, and general expressions are not included.

(13) A limited form of OPEN statement is included with the following olist specifiers required, and no others are allowed:

(a) An integer constant unit identifier

(b) The keyword specifier ACCESS= 'DIRECT'

(c) The record length specifier RECL= rl, where rl is an integer constant

The OPEN statement is included in the subset only to the extent needed to connect a unit to a direct access unformatted file. Once a unit has been connected to a direct access file, it may not be reconnected to any other file.

(14) Named files are not included.

D3.13 Section 13: Format Specification. The subset is the same as the full language except that:

(1) The following edit descriptors are not included:

Iw.m	Tc	S
Dw.d	TLc	SP
Gw.d	TRc	SS
Gw.dEe		

(2) At most three levels of parentheses are permitted.

(3) The format scan terminator (colon) is not included.

D3.14 Section 14: Main Program. The subset is the same as the full language.

D3.15 Section 15: Functions and Subroutines. The subset is the same as the full language except that the following are not included:

(1) The ENTRY statement

(2) Alternate return specifier

(3) Generic function references

(4) Intrinsic functions involving arguments or results of type double precision or complex

Other exclusions are presented in D3.19, most notably an asterisk character length specifier, character functions, the intrinsic functions LEN, CHAR, and INDEX, and partial association.

D3.16 Section 16: Block Data Subprogram. Block data subprograms are not included in the subset.

D3.17 Section 17: Association and Definition. The subset is the same as the full language except that the concept of partial association does not apply to the subset.

D3.18 Section 18: Scope and Classes of Symbolic Names. The subset is the same as the full language.

D3.19 Sections 1 to 18: Character Type. The primary intent of the the subset character facility is to provide a minimal character capability that is functionally comparable to what is possible with most extensions of Hollerith data.

D3.19.1 Character Features in the Subset. The subset includes the following character data type features:

(1) Character constants, variables, and arrays, but not character functions

(2) CHARACTER and IMPLICIT statements for declaring character entities and their lengths; a length specification must be an integer constant (not an asterisk)

(3) Character assignment statements in which the right-hand side is a variable, array element, or constant

(4) Character relational expressions in which the operands are variables, array elements, or constants

(5) Initialization of character variables, arrays, and array elements in a DATA statement

(6) Character variables, arrays, and array elements in output lists

(7) Character variables, arrays, array elements, and constants as arguments in subprogram references

(8) Character constants (but not variables or array elements) as a format specification

(9) Total, but not partial, association of character entities (that is, association of character entities only of the same length by means of COMMON and EQUIVALENCE statements or by argument association)

(10) Input/output of character data, both formatted (using character edit descriptors) and unformatted

D3.19.2 <u>Character Features Not in the Subset</u>. The subset does not include the following character data type features:

(1) Substring reference and definition

(2) Concatenation operator

(3) Use of character variables or array elements as format specifications

(4) Partial association of character entities

(5) Character functions

(6) The intrinsic functions LEN, CHAR, and INDEX

(7) Character length specification consisting of an asterisk or any expression other than a constant

D4 <u>Subset Conformance</u>

Conformance at the subset level of this standard involves requirements that relate to the full language for both processors and programs.

D4.1 <u>Subset Processor Conformance</u>. A standard-conforming subset processor may include an extension to the subset language that has an interpretation in the full language only if the processor provides the interpretation described for the full language. That is, a standard-conforming subset processor may not provide an extension that conflicts with the full language. Extensions that do not have forms and interpretations in the full language are not precluded by this requirement.

As an example, a standard-conforming subset processor may provide a double precision data type provided that the requirements for double precision are fulfilled.

D4.2 <u>Subset Program Conformance</u>. A program that conforms to the subset level of this standard must have the same interpretation at both the subset level and the full language level. The principal implication of this requirement concerns the use of function names that are identified as specific or generic intrinsic function names at the full language level but which are not available at the subset level. Examples of such names are DSIN, MIN, and CABS.

A subset-conforming program may not use such names as intrinsic functions because these names are not defined as intrinsic functions in the subset language. Moreover, a subset-conforming program may not use such names as external function names unless such names are identified as external function names by appearing in an EXTERNAL statement. If such names are not explicitly declared as external, the names would be classified as external by a subset processor and as intrinsic by a full language processor. Note that the burden of avoiding this situation rests on the program. A subset-conforming processor is not required to recognize that a full language intrinsic name is being used without being declared as external. In effect, the full set of names described in Table 5 may be considered as reserved intrinsic function names in the subset even though only a subset of those names is available for use.

APPENDIX E: FORTRAN STATEMENTS

Form	Descriptive Heading
ASSIGN s TO i	Statement Label Assignment Statement
BACKSPACE u	File Positioning Statement
CALL sub [([a [,a]...])]	Subroutine Reference: CALL Statement
CHARACTER [*len[,]] nam [,nam]...	Character Type-Statement
COMMON [/[cb]/]nlist[[,]/[cb]/nlist]...	COMMON Statement
CONTINUE	CONTINUE Statement
DATA nlist/clist/ [[,]nlist/clist/]...	DATA Statement
DIMENSION a(d) [,a(d)]...	DIMENSION Statement
DO s [,] i=e_1,e_2[,e_3]	DO Statement
ELSE	ELSE Statement
ELSE IF (e) THEN	ELSE IF Statement
END	END Statement
END IF	END IF Statement
ENDFILE u	File Positioning Statement
EQUIVALENCE (nlist) [,(nlist)]...	EQUIVALENCE Statement
EXTERNAL proc [,proc]...	EXTERNAL Statement
FORMAT fs	FORMAT Statement

APPENDIX E: FORTRAN STATEMENTS

Form	Descriptive Heading
ASSIGN s TO i	Statement Label Assignment Statement
BACKSPACE u BACKSPACE (alist)	File Positioning Statements
BLOCK DATA [sub]	BLOCK DATA Statement
CALL sub [([a [,a]...])]	Subroutine Reference: CALL Statement
CHARACTER [*len[,]] nam [,nam]...	Character Type-Statement
CLOSE (cllist)	CLOSE Statement
COMMON [/[cb]/]nlist[[,]/[cb]/nlist]...	COMMON Statement
COMPLEX v [,v]...	Complex Type-Statement
CONTINUE	CONTINUE Statement
DATA nlist/clist/ [[,]nlist/clist/]...	DATA Statement
DIMENSION a(d) [,a(d)]...	DIMENSION Statement
DO s [,] i=e_1,e_2[,e_3]	DO Statement
DOUBLE PRECISION v [,v]...	Double Precision Type-Statement
ELSE	ELSE Statement
ELSE IF (e) THEN	ELSE IF Statement
END	END Statement
END IF	END IF Statement
ENDFILE u ENDFILE (alist)	File Positioning Statements
ENTRY en [([d [,d]...])]	ENTRY Statement
EQUIVALENCE (nlist) [,(nlist)]...	EQUIVALENCE Statement
EXTERNAL proc [,proc]...	EXTERNAL Statement
FORMAT fs	FORMAT Statement

Form	Descriptive Heading
fun ([d [,d]...]) = e	Statement Function Statement
[typ] FUNCTION fun ([d [,d]...])	FUNCTION Statement
GO TO i [[,](s [,s]...)]	Assigned GO TO Statement
GO TO s	Unconditional GO TO Statement
GO TO (s [,s]...)[,] i	Computed GO TO Statement
IF (e) st	Logical IF Statement
IF (e) s_1, s_2, s_3	Arithmetic IF Statement
IF (e) THEN	Block IF Statement
IMPLICIT typ (a [,a]...) [,typ (a [,a]...)]...	IMPLICIT Statement
INTEGER v [,v]...	Integer Type-Statement
INTRINSIC fun [,fun]...	INTRINSIC Statement
LOGICAL v [,v]...	Logical Type-Statement
OPEN (olist)	OPEN Statement
PAUSE [n]	PAUSE Statement
PROGRAM pgm	PROGRAM Statement
READ (cilist) [iolist]	Data Transfer Input Statement

Form	Descriptive Heading
fun ([d [,d]...]) = e	Statement Function Statement
[typ] FUNCTION fun ([d [,d]...])	FUNCTION Statement
GO TO i [[,](s [,s]...)]	Assigned GO TO Statement
GO TO s	Unconditional GO TO Statement
GO TO (s [,s]...)[,] i	Computed GO TO Statement
IF (e) st	Logical IF Statement
IF (e) s_1, s_2, s_3	Arithmetic IF Statement
IF (e) THEN	Block IF Statement
IMPLICIT typ (a [,a]...) [,typ (a [,a]...)]...	IMPLICIT Statement
INQUIRE (iflist)	INQUIRE by File Statement
INQUIRE (iulist)	INQUIRE by Unit Statement
INTEGER v [,v]...	Integer Type-Statement
INTRINSIC fun [,fun]...	INTRINSIC Statement
LOGICAL v [,v]...	Logical Type-Statement
OPEN (olist)	OPEN Statement
PARAMETER (p=e [,p=e]...)	PARAMETER Statement
PAUSE [n]	PAUSE Statement
PRINT f [,iolist]	Data Transfer Output Statement
PROGRAM pgm	PROGRAM Statement
READ (cilist) [iolist]	Data Transfer Input Statement
READ f [,iolist]	Data Transfer Input Statement

Form	Descriptive Heading
REAL v [,v]...	Real Type-Statement
RETURN	RETURN Statement
REWIND u	File Positioning Statement
SAVE a [,a]...	SAVE Statement
STOP [n]	STOP Statement
SUBROUTINE sub [([d [,d]...])]	Subroutine Subprogram and SUBROUTINE Statement
v = e	Arithmetic Assignment Statement
v = e	Logical Assignment Statement
v = e	Character Assignment Statement
WRITE (cilist) [iolist]	Data Transfer Output Statement

Form	Descriptive Heading
REAL v [,v]...	Real Type-Statement
RETURN [e]	RETURN Statement
REWIND u REWIND (alist)	File Positioning Statements
SAVE [a [,a]...]	SAVE Statement
STOP [n]	STOP Statement
SUBROUTINE sub [([d [,d]...])]	Subroutine Subprogram and SUBROUTINE Statement
v = e	Arithmetic Assignment Statement
v = e	Logical Assignment Statement
v = e	Character Assignment Statement
WRITE (cilist) [iolist]	Data Transfer Output Statement

APPENDIX F: SYNTAX CHARTS

The charts in this Appendix describe the syntax of the FORTRAN language as specified in this standard.

The charts have been designed for human readability, not as a basis for parsing. For example, the description of expressions does not reflect the precedence of operators. Certain syntactic features are not represented in the charts. These include:

(1) Use of blanks.

(2) The manner of writing statements on initial lines and continuation lines.

(3) Comment lines.

(4) Context-dependent features, such as data type requirements, uniqueness and completeness of labels used, actual and dummy argument matching, requirements for specification statements, restrictions on the use of statements in a particular context, etc. Some restrictions of this kind are given in footnotes.

If there is a discrepancy between the syntax charts of this Appendix and the language as specified in the standard, the language syntax is that specified by the standard.

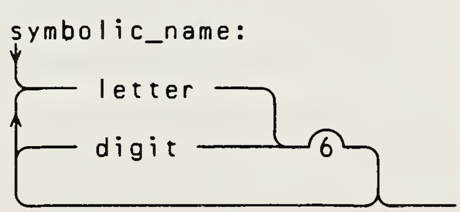
F1 Chart Conventions

In the charts, sequences of lowercase letters and embedded underscore characters (_) represent syntactic entities. Uppercase letters and special characters must appear as written.

In general, names of syntactic items are identical to those used in the standard. A few names have been shortened (for example, "statement label" to "label").

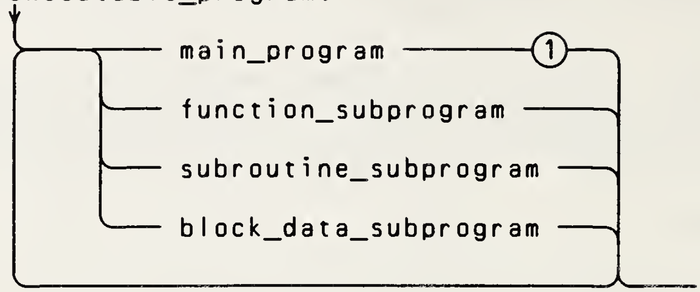
The charts are in the form of a "railroad track" (hence the term "railroad normal form"). Alternative paths are specified by "switches" in the path. A number _n_ in a half-circle indicates that the path may be traversed at most _n_ times. A number _n_ in a circle indicates that the path must be traversed exactly _n_ times.

For example, a symbolic name takes the form of one to six letters or digits, the first of which must be a letter. The syntax chart for a symbolic name is:



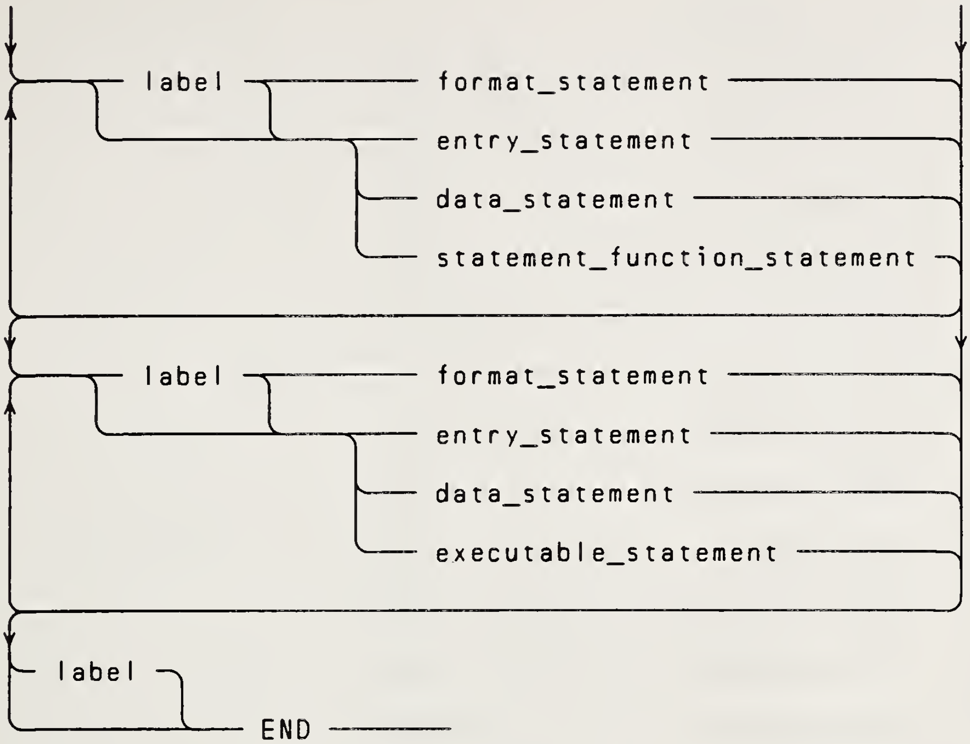
F2 Charts

1 executable_program:



(1) An executable program must contain one and only one main program.

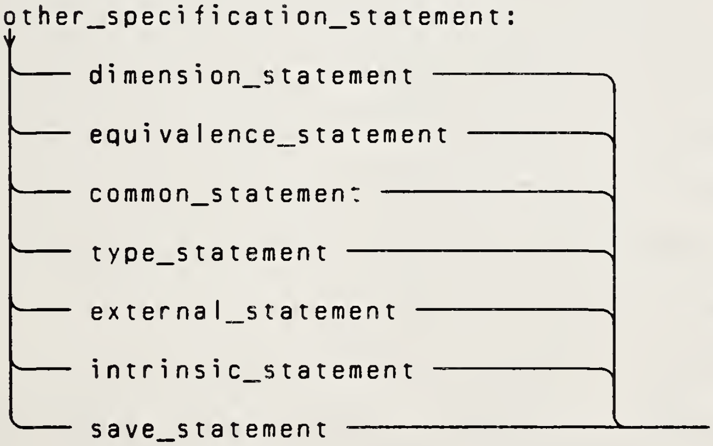
An executable program may contain external procedures specified by means other than FORTRAN.



(2) A main program may not contain an ENTRY or RETURN statement.

(5) A block data subprogram may contain only BLOCK DATA, IMPLICIT, PARAMETER, DIMENSION, COMMON, SAVE, EQUIVALENCE, DATA, END, and type-statements.

6 other_specification_statement:

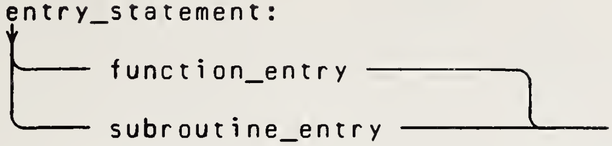


```
7   executable_statement:
      ├── assignment_statement ──────┐
      ├── goto_statement ────────────┤
      ├── arithmetic_if_statement ───┤
      ├── logical_if_statement ──────┤
      ├── block_if_statement ────────┤
      ├── else_if_statement ─────────┤
      ├── else_statement ────────────┤
      ├── end_if_statement ──────────┤
      ├── do_statement ──────────────┤
      ├── continue_statement ────────┤
      ├── stop_statement ────────────┤
      ├── pause_statement ───────────┤
      ├── read_statement ────────────┤
      ├── write_statement ───────────┤
      ├── print_statement ───────────┤
      ├── rewind_statement ──────────┤
      ├── backspace_statement ───────┤
      ├── endfile_statement ─────────┤
      ├── open_statement ────────────┤
      ├── close_statement ───────────┤
      ├── inquire_statement ─────────┤
      ├── call_statement ────────────┤
      └── return_statement ──────────┴────
```

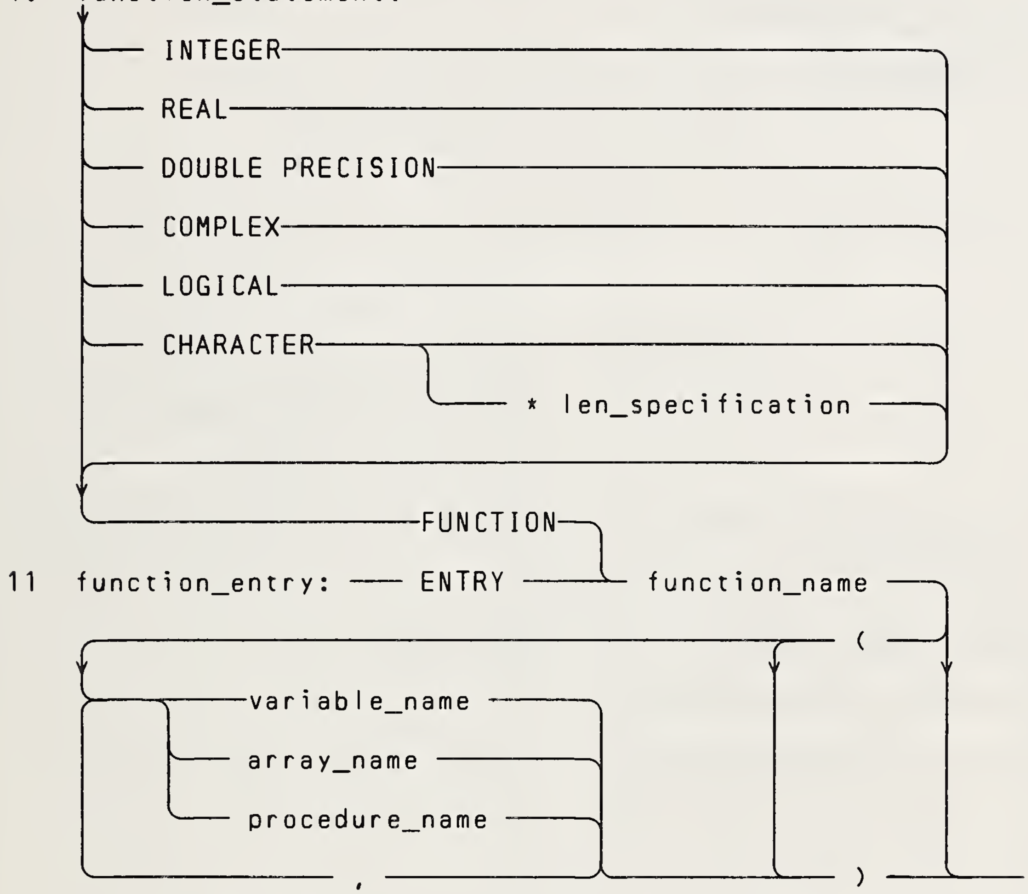
(7) An END statement is also an executable statement and must appear as the last statement of a program unit.

```
8   program_statement: ──── PROGRAM program_name ────
```

9 entry_statement:

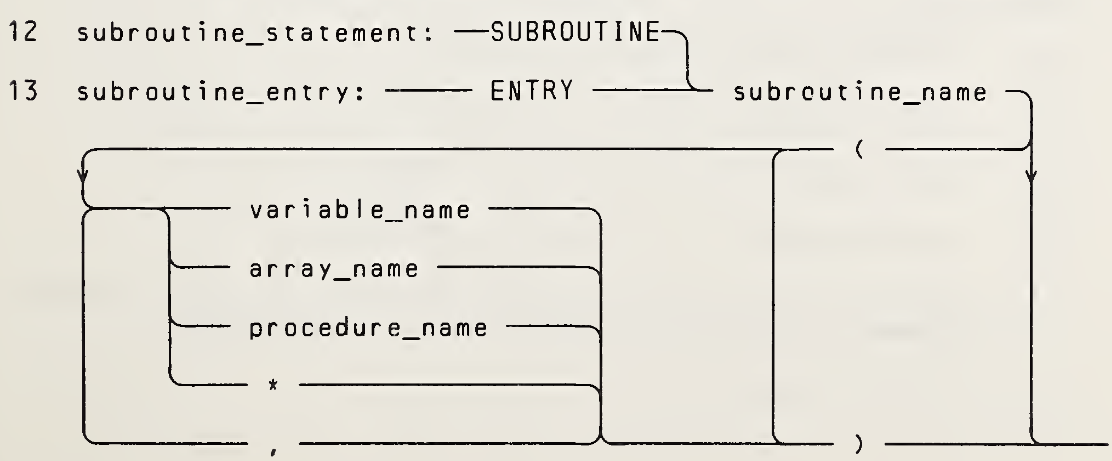


10 function_statement:



(11) The parentheses must appear in a FUNCTION statement.

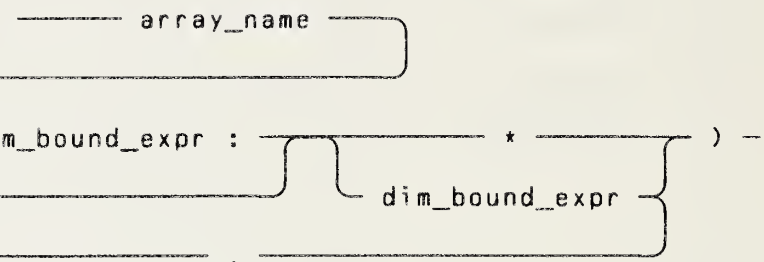
12 subroutine_statement:



14 block_data_statement:

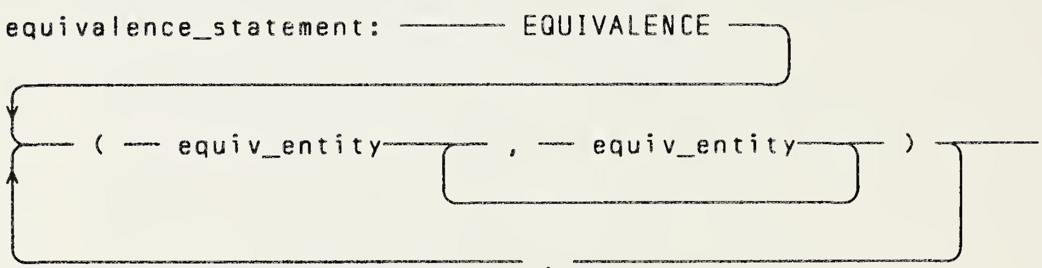
15 dimension_statement:

16 array_declarator:

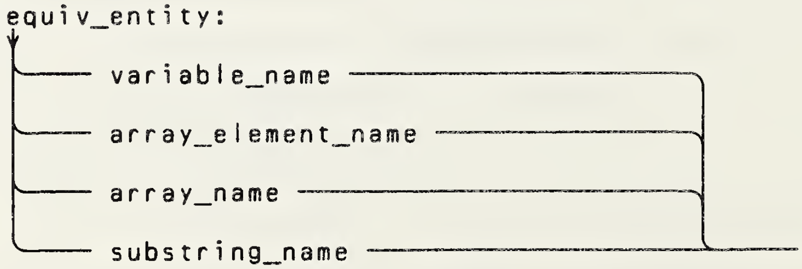


(16) Only a dummy array declarator (5.1.2.2) may contain an asterisk.

17 equivalence_statement:

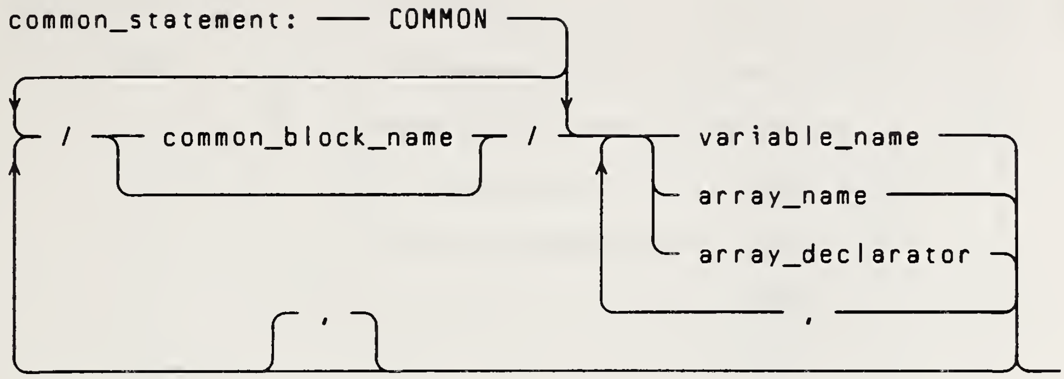


18 equiv_entity:

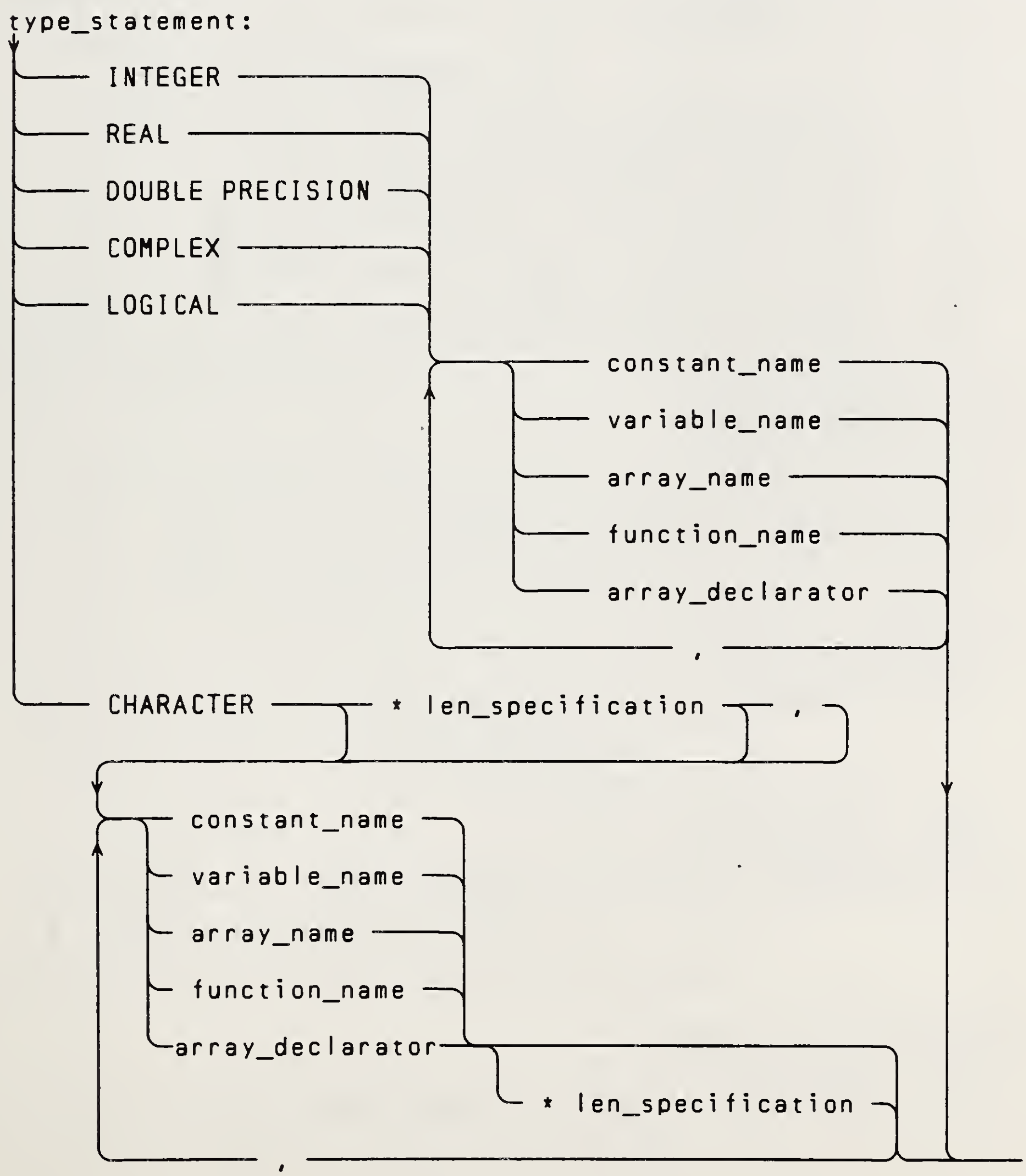


(18) A subscript or substring expression in an EQUIVALENCE statement must be an integer constant expression.

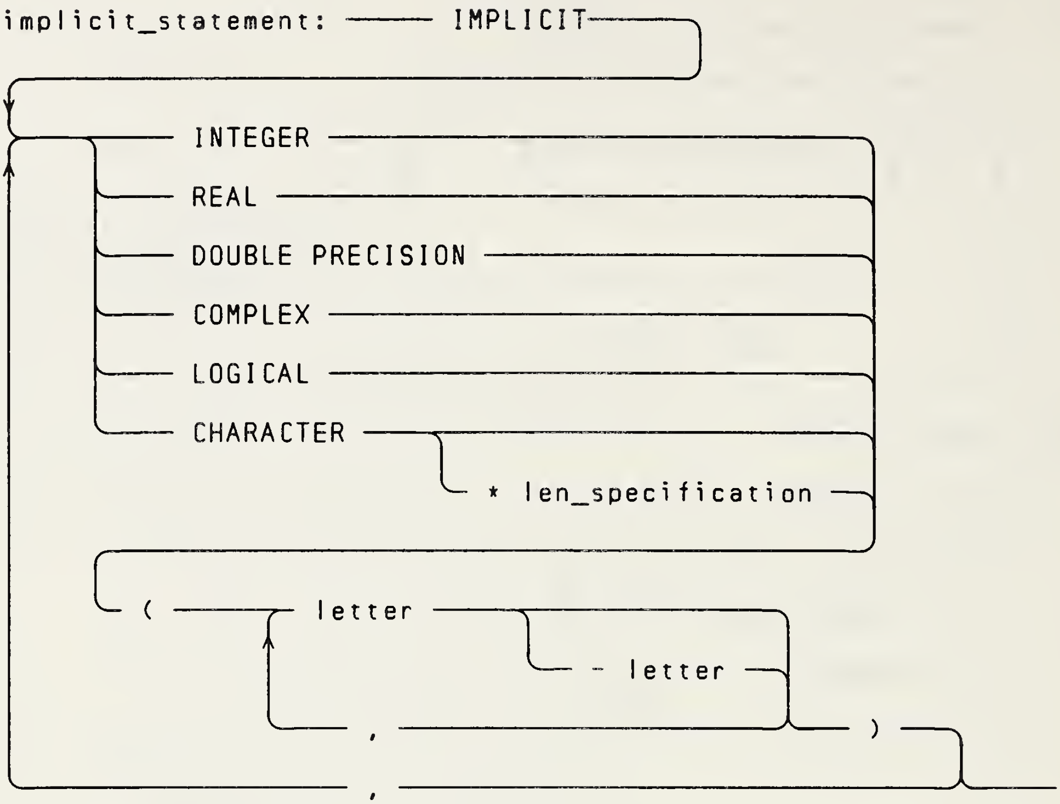
19


common_statement:
COMMON
/
common_block_name
/
variable_name
array_name
array_declarator
,
,

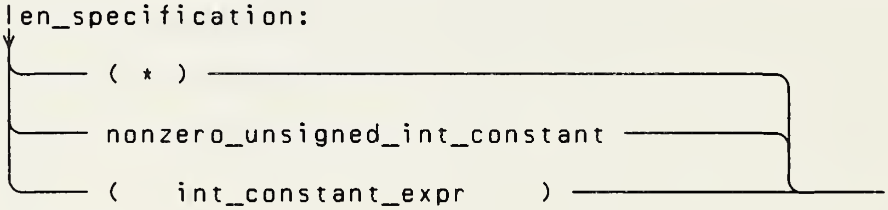
20


type_statement:
INTEGER
REAL
DOUBLE PRECISION
COMPLEX
LOGICAL
constant_name
variable_name
array_name
function_name
array_declarator
,
CHARACTER
* len_specification
,
constant_name
variable_name
array_name
function_name
array_declarator
* len_specification
,

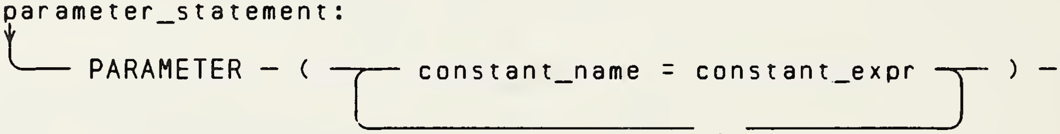
21 implicit_statement:



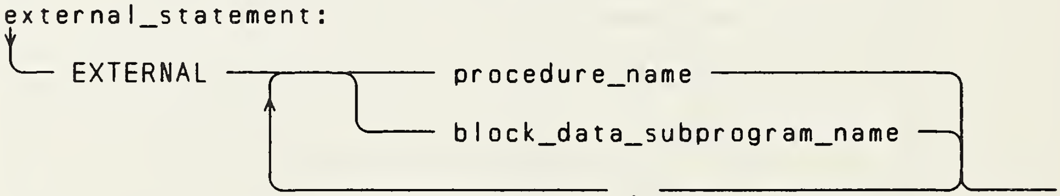
22 len_specification:



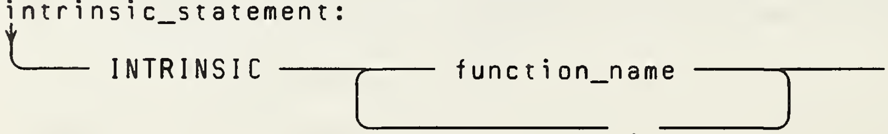
23 parameter_statement:



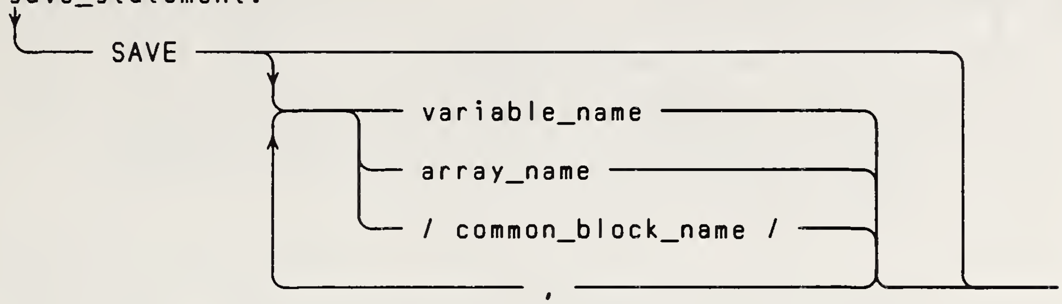
24 external_statement:



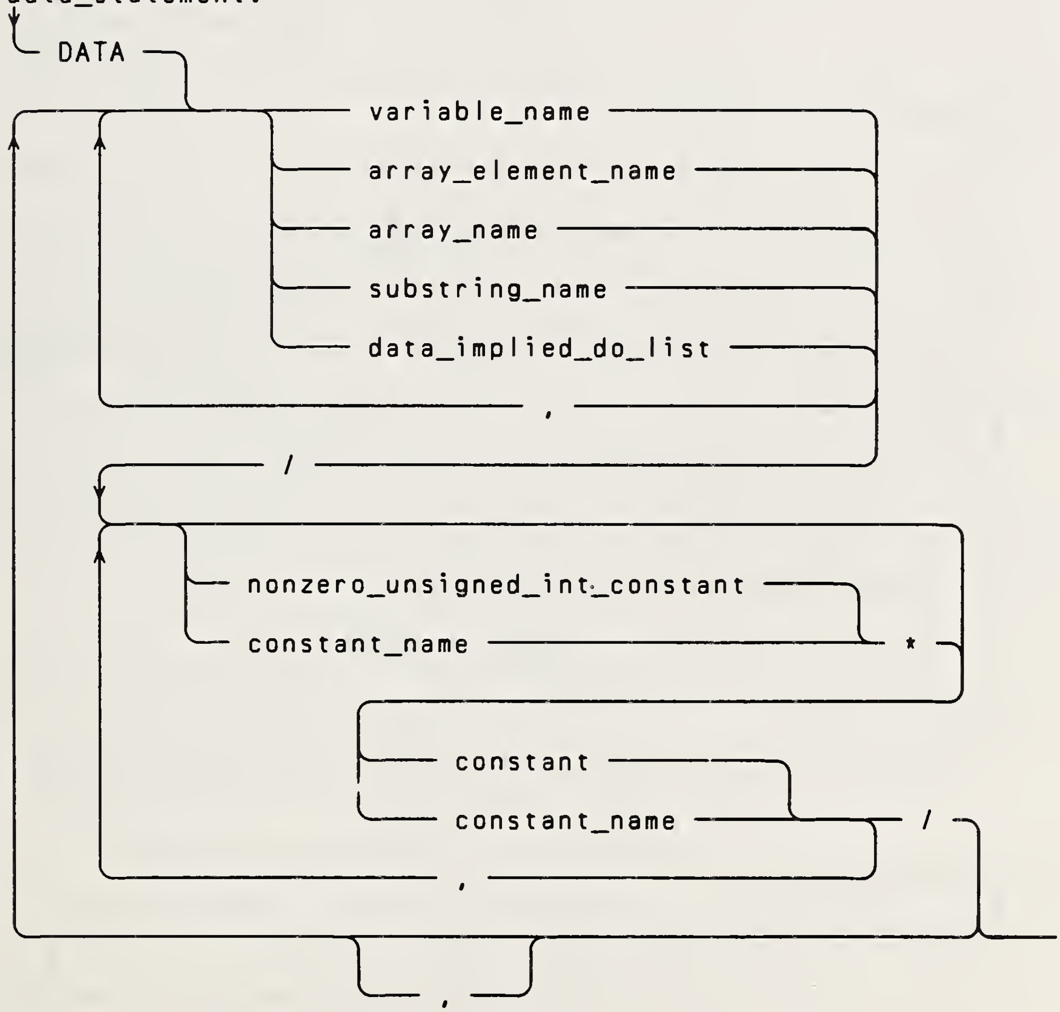
25 intrinsic_statement:



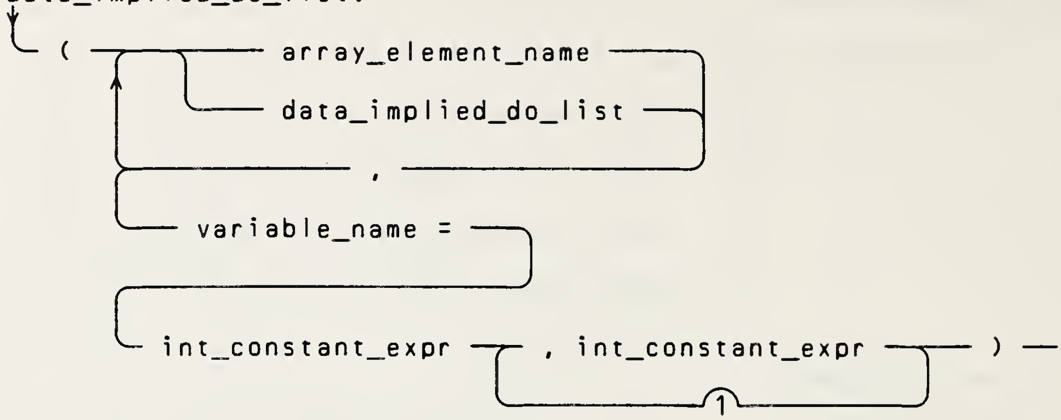
26 save_statement:



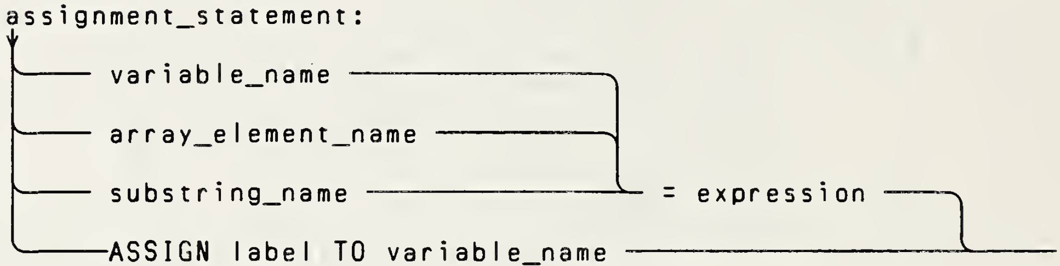
27 data_statement:



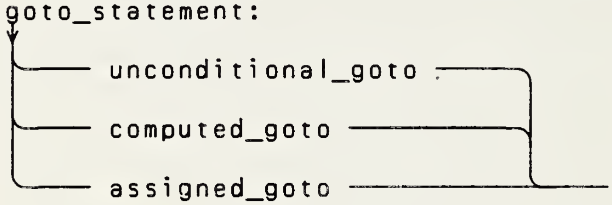
28 data_implied_do_list:



29 assignment_statement:

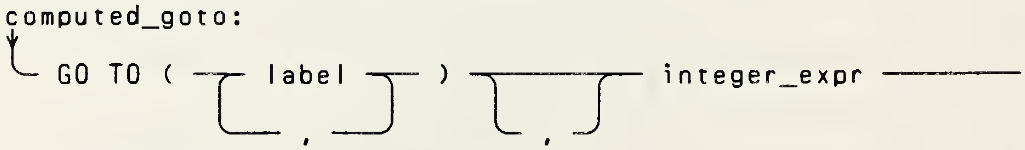


30 goto_statement:

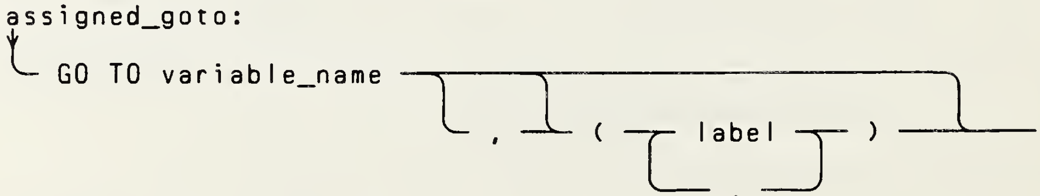


31 unconditional_goto: —— GO TO —— label ——

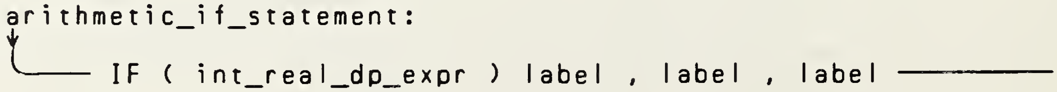
32 computed_goto:



33 assigned_goto:



34 arithmetic_if_statement:



35 logical_if_statement:

```
IF ( logical_expression ) executable_statement
```

(35) The executable statement contained in a logical IF statement must not be a DO, block IF, ELSE IF, ELSE, END IF, END, or another logical IF statement.

36 block_if_statement:

```
IF ( logical_expression ) THEN
```

37 else_if_statement:

```
ELSE IF ( logical_expression ) THEN
```

38 else_statement: ELSE

39 end_if_statement: END IF

40 do_statement:

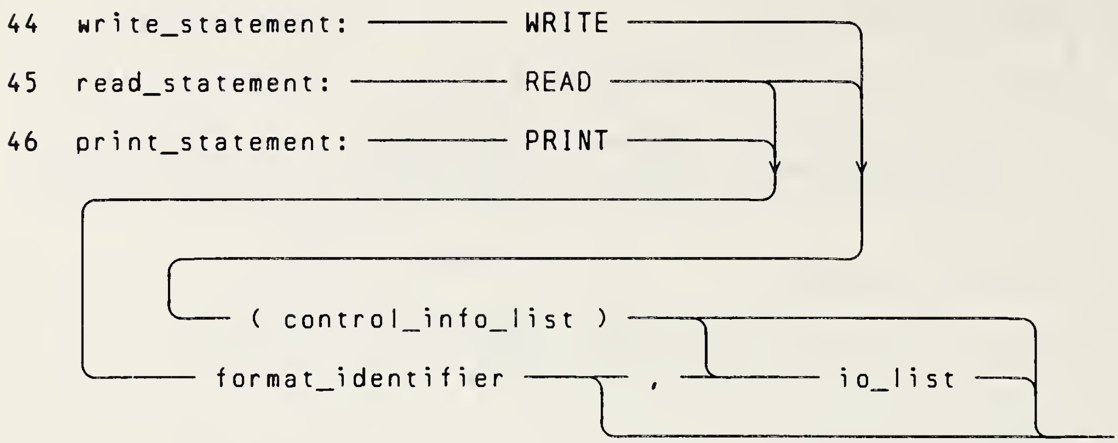
```
DO label ,
variable_name = int_real_dp_expr , int_real_dp_expr
                                   1
```

41 continue_statement: CONTINUE

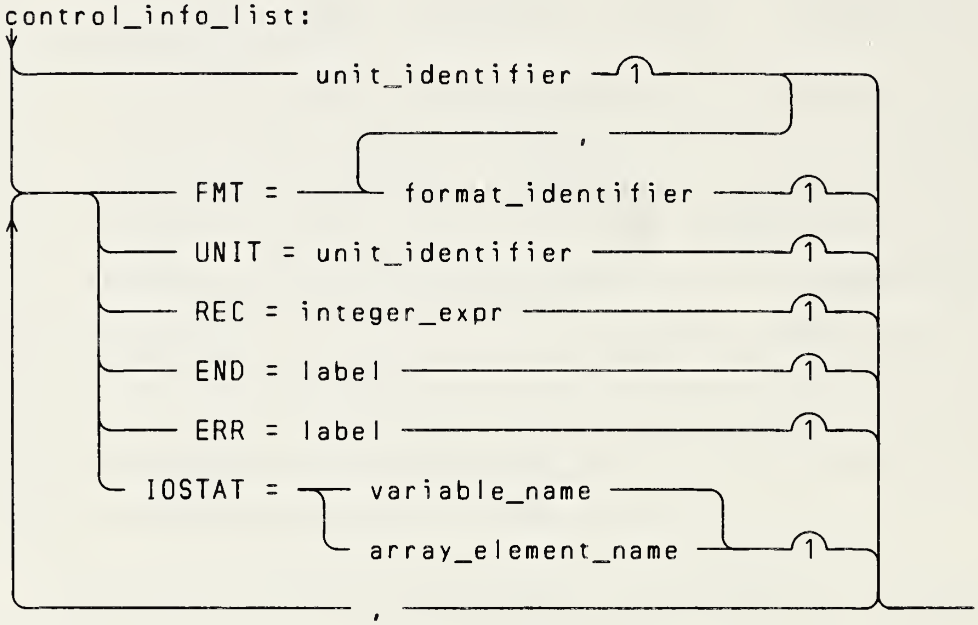
42 stop_statement: STOP

43 pause_statement: PAUSE

```
digit
  5
character_constant
```

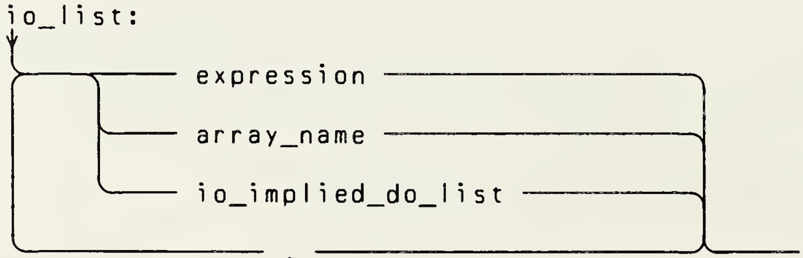


47 control_info_list:



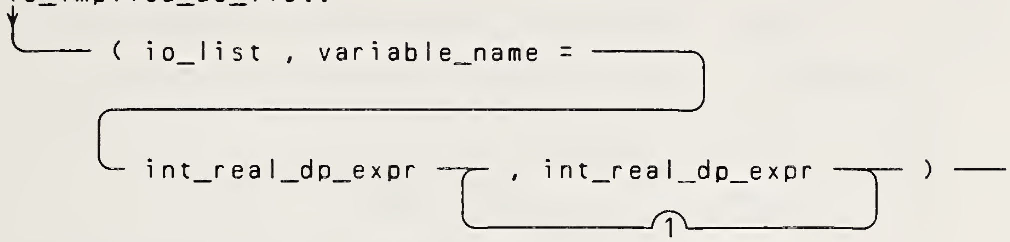
(47) A control_info_list must contain exactly one unit_identifier. An END= specifier must not appear in a WRITE statement.

48 io_list:

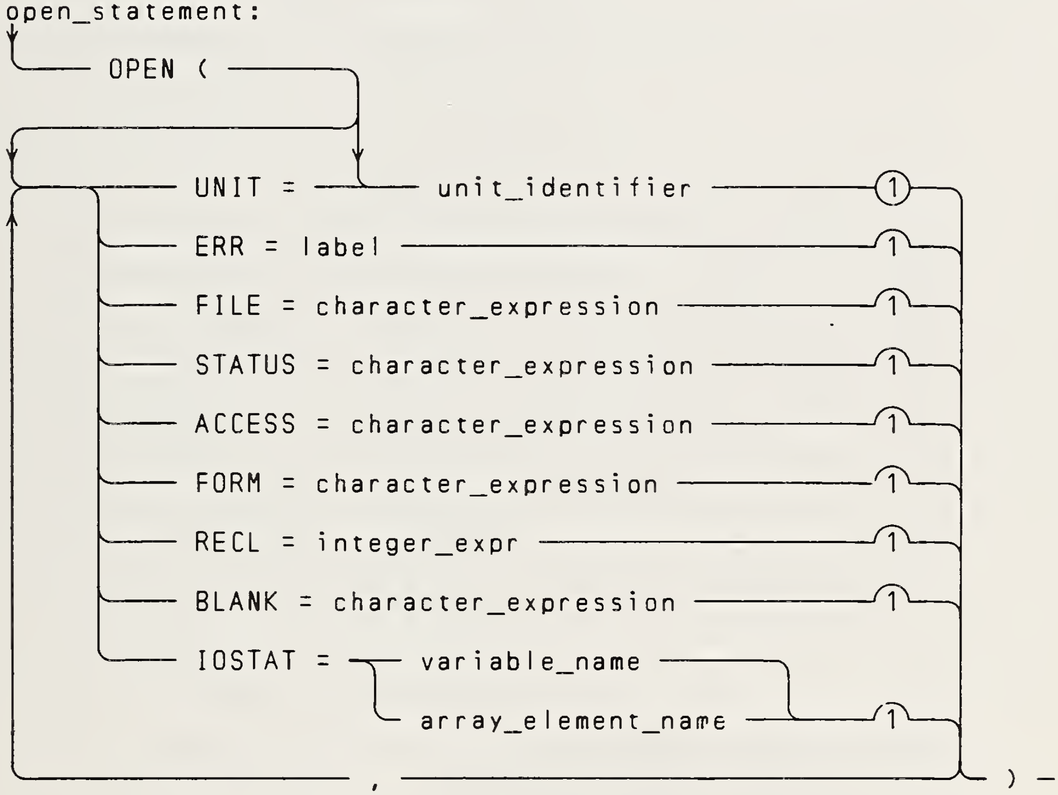


(48) In a READ statement, an input/output list expression must be a variable name, array element name, or substring name.

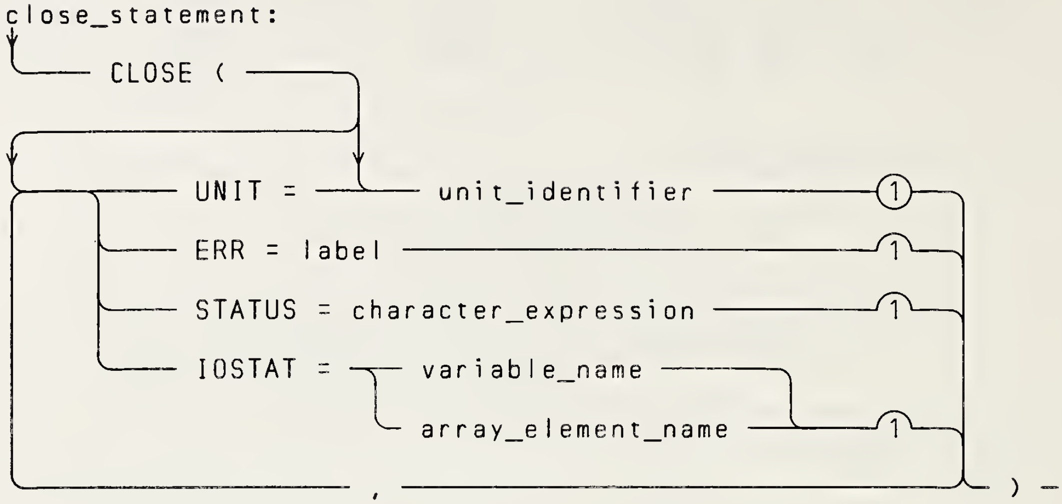
49 io_implied_do_list:



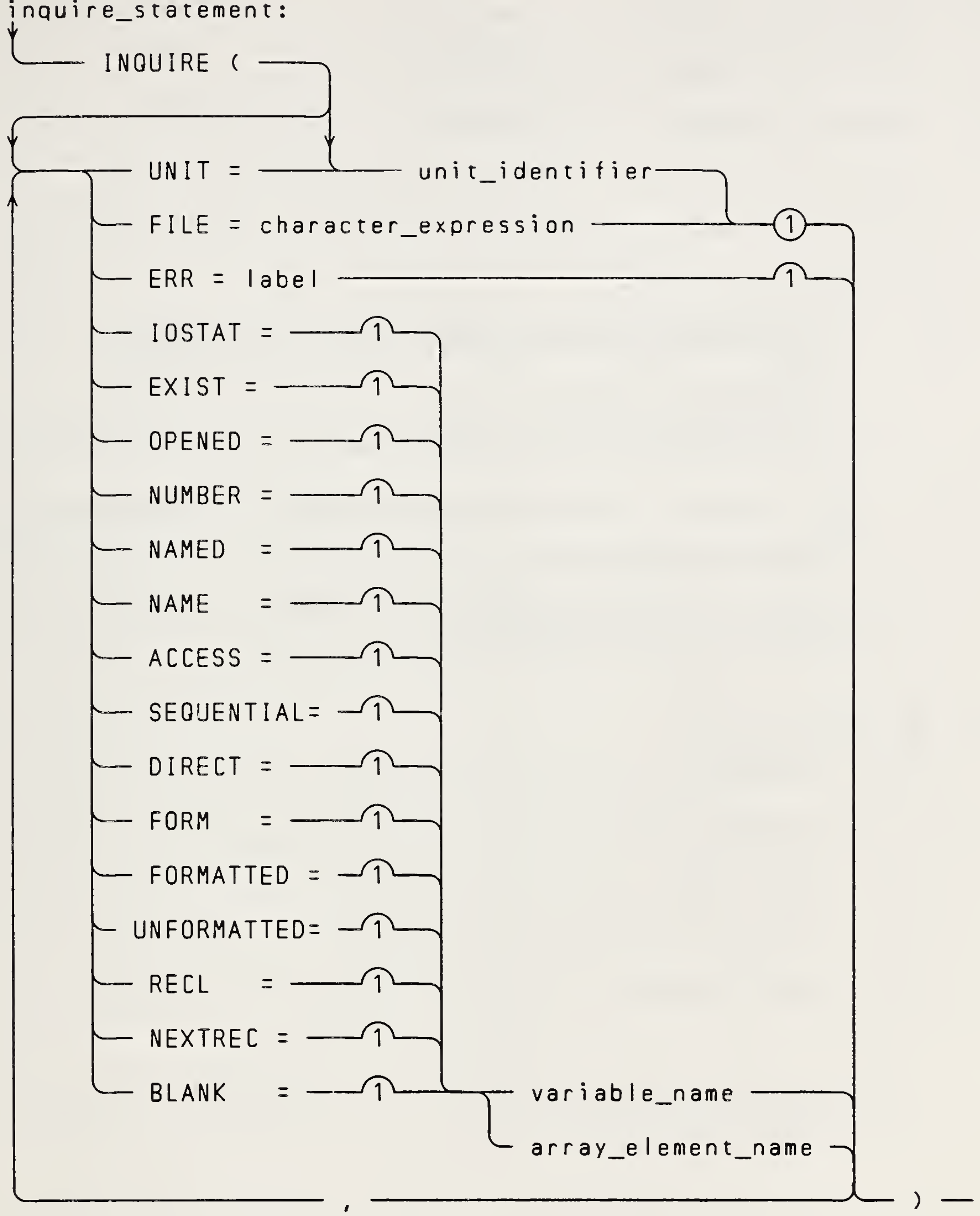
50 open_statement:

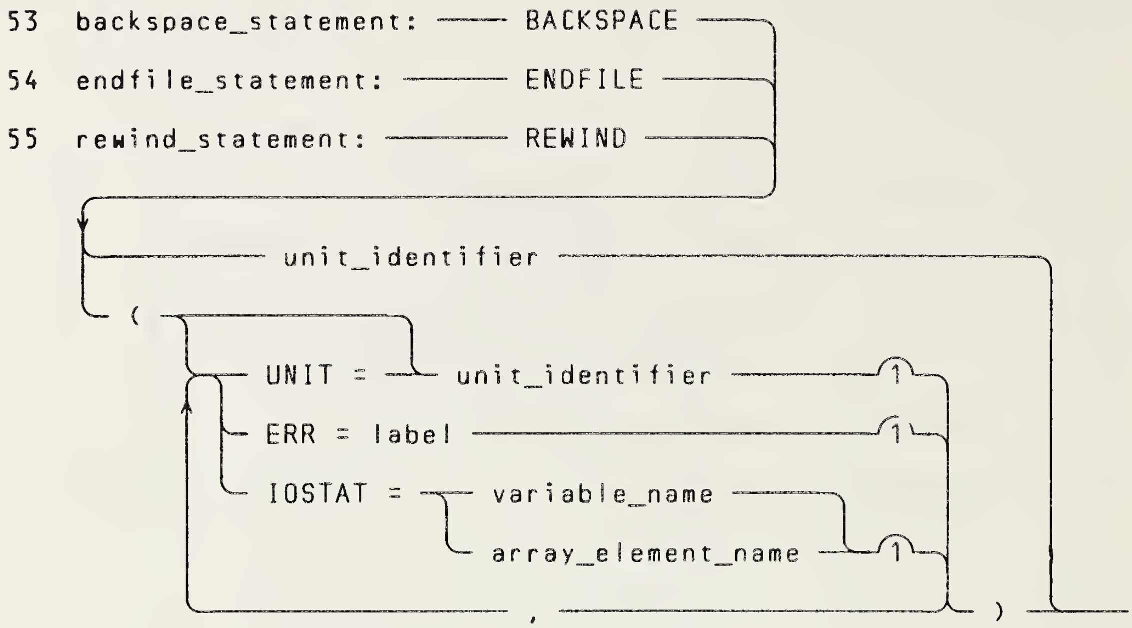


51 close_statement:

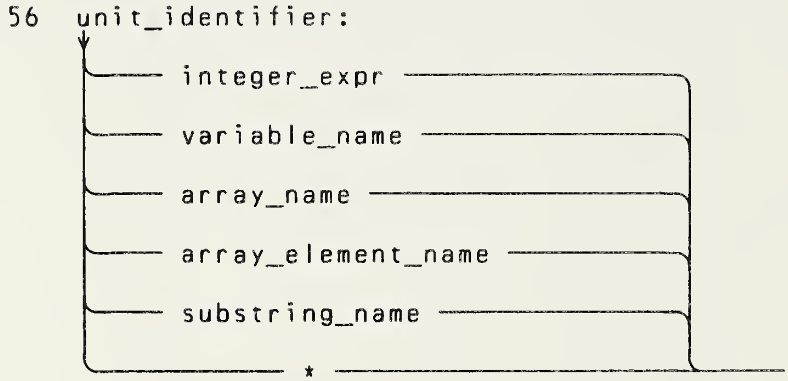


52 inquire_statement:



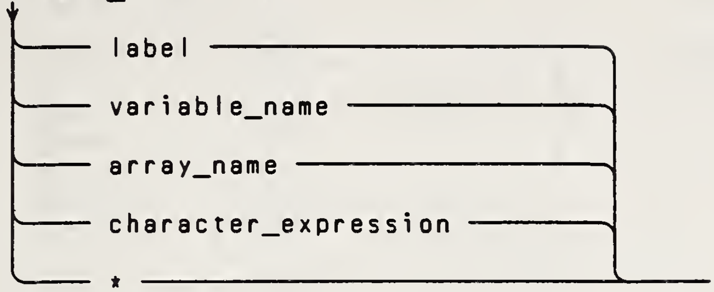


(53,54,55) BACKSPACE, ENDFILE, and REWIND statements must contain a unit identifier.



(56) An unit identifier must be of type integer or character, or be an asterisk.

57 format_identifier:

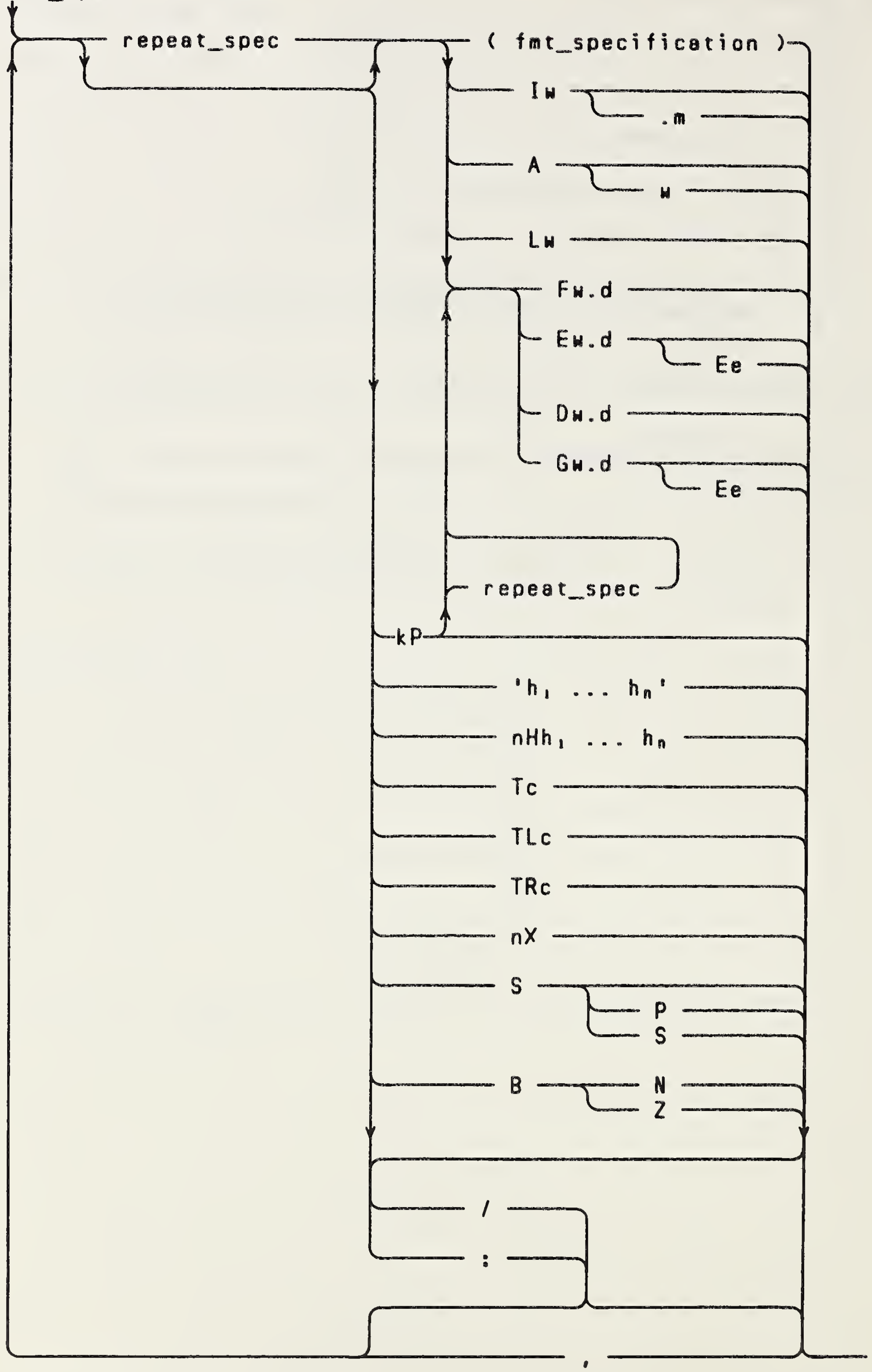


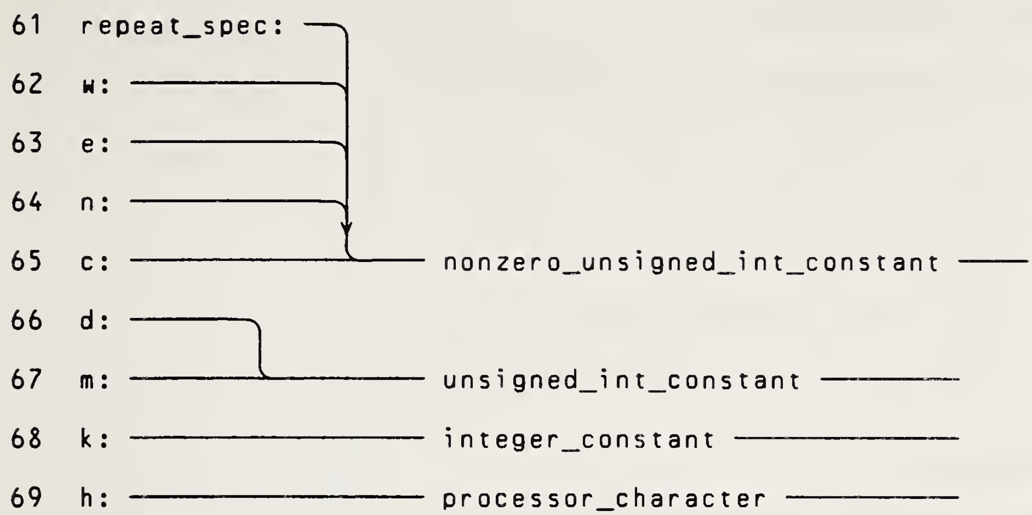
(57) A format identifier that is a variable name or array name must be of type integer or character.

58 format_statement: ——— FORMAT format_specification ———

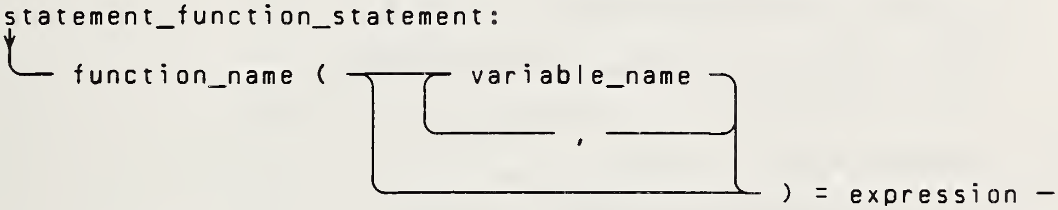
59 format_specification:

60 fmt_specification:

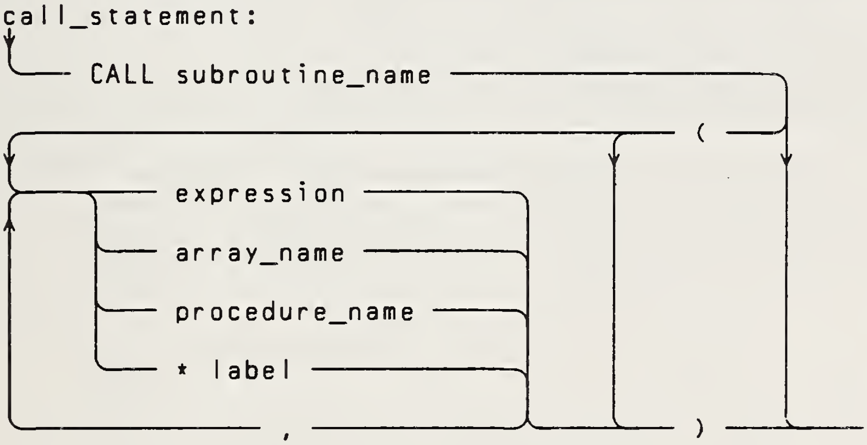




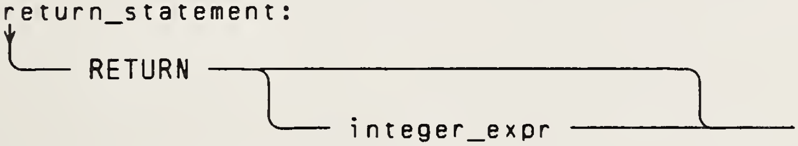
70 statement_function_statement:



71 call_statement:

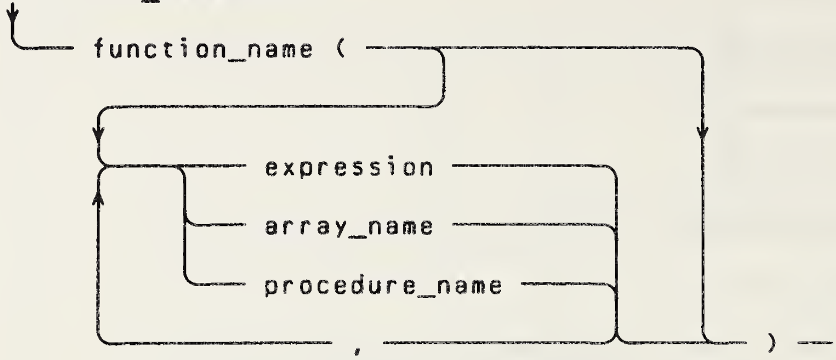


72 return_statement:

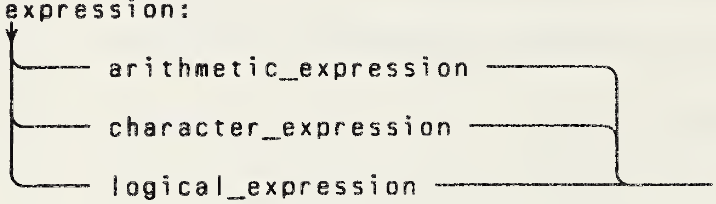


(72) An alternate return is not allowed in a function subprogram.

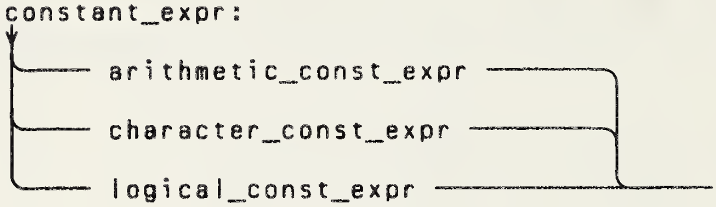
73 function_reference:

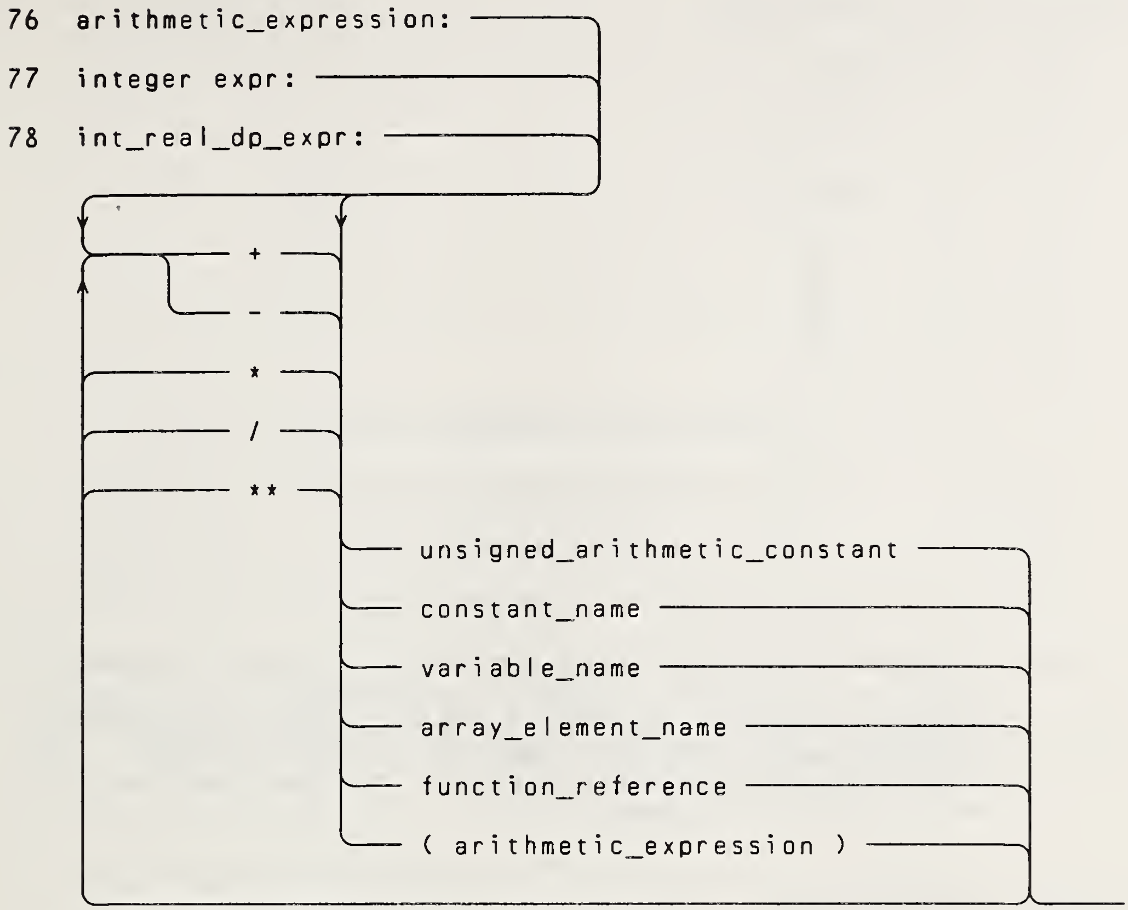


74 expression:



75 constant_expr:



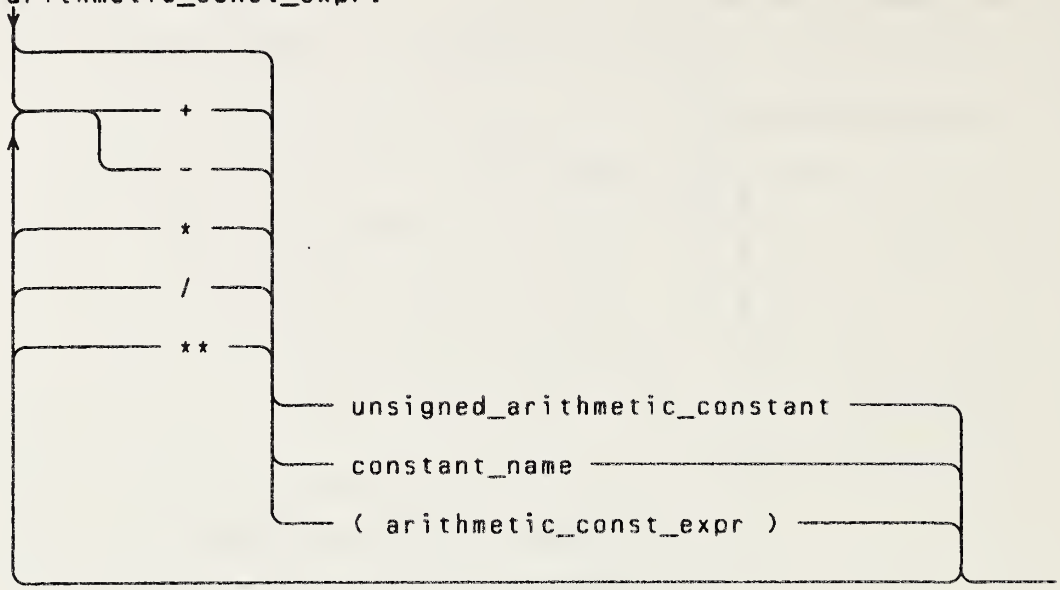


(76) A constant name, variable name, array element name, or function reference in an arithmetic expression must be of type integer, real, double precision, or complex. Tables 2 and 3 (6.1.4) list prohibited combinations involving operands of type complex.

(77) An integer expression is an arithmetic expression of type integer.

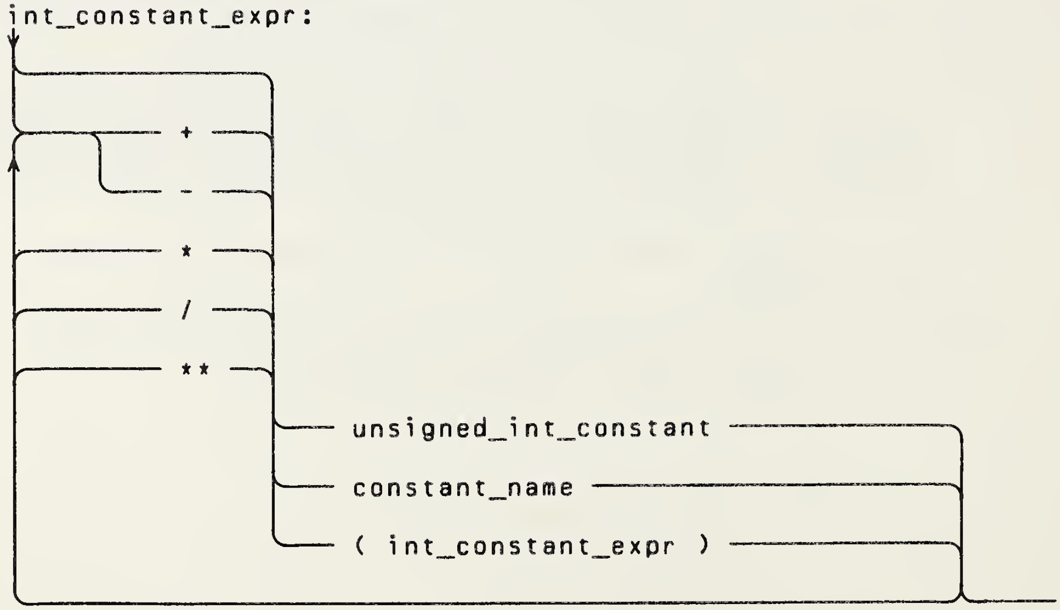
(78) An int_real_dp_expression is an arithmetic expression of type integer, real, or double precision.

79 arithmetic_const_expr:



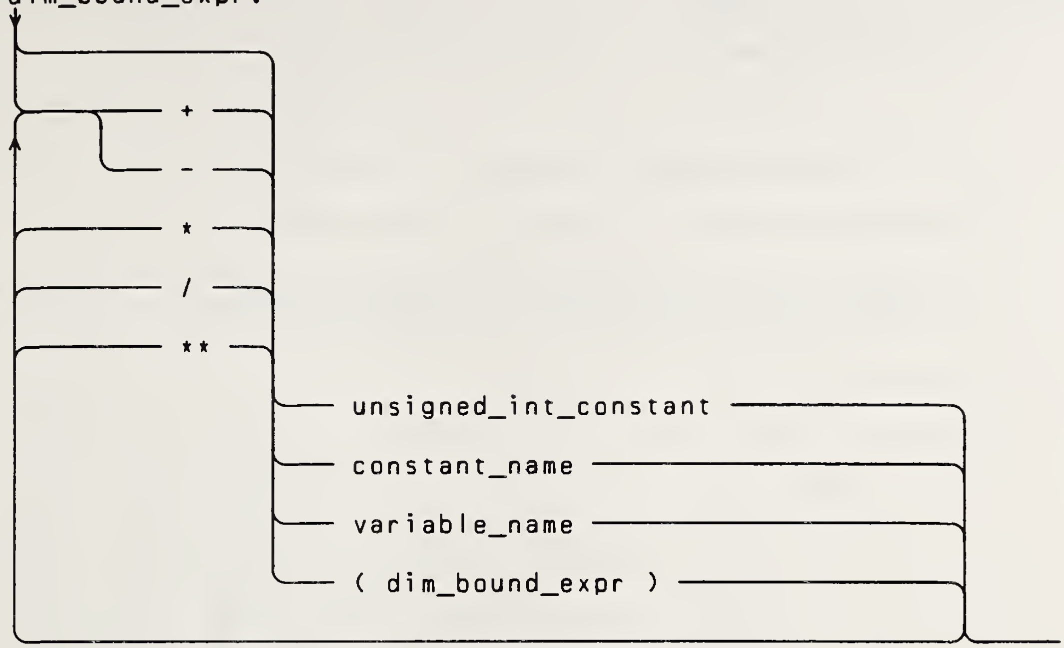
(79) A constant name in an arithmetic constant expression must be of type integer, real, double precision, or complex. Tables 2 and 3 (6.1.4) list prohibited combinations involving operands of type complex. The right hand operand (the exponent) of the ** operator must be of type integer.

80 int_constant_expr:



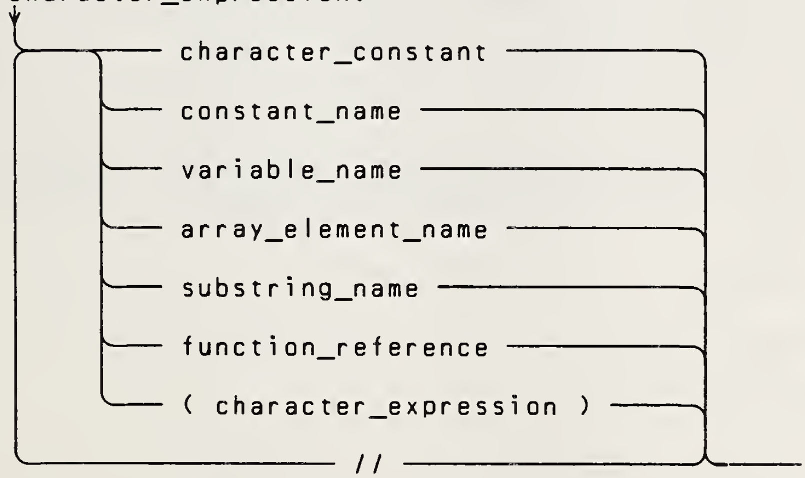
(80) A constant name in an integer constant expression must be of type integer.

81 dim_bound_expr:



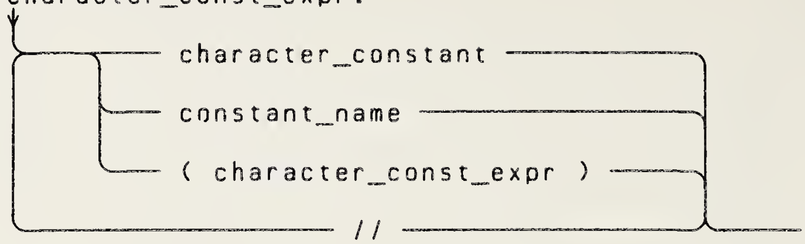
(81) Each variable name in a dimension bound expression must be of type integer and must be a dummy argument or in a common block.

82 character_expression:



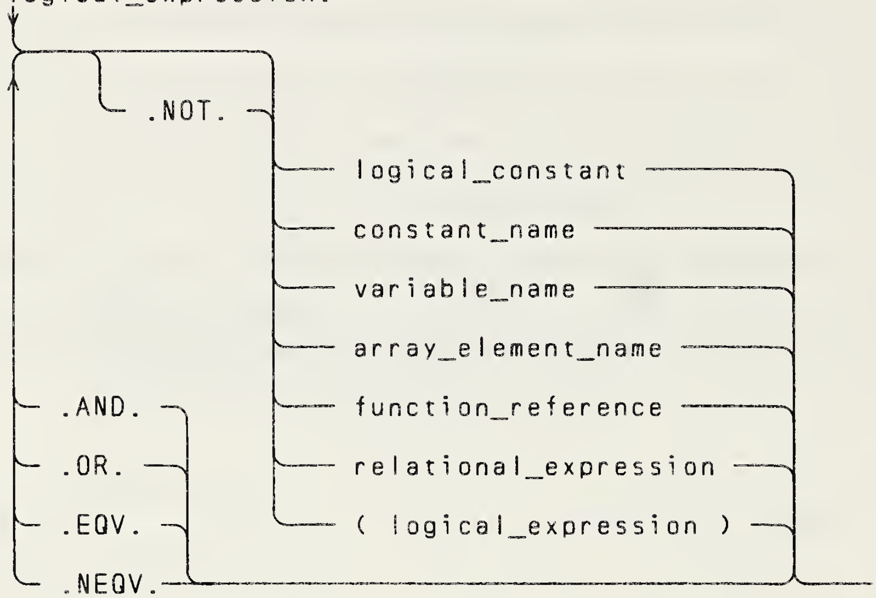
(82) A constant name, variable name, array element name, or function reference must be of type character in a character expression.

83 character_const_expr:



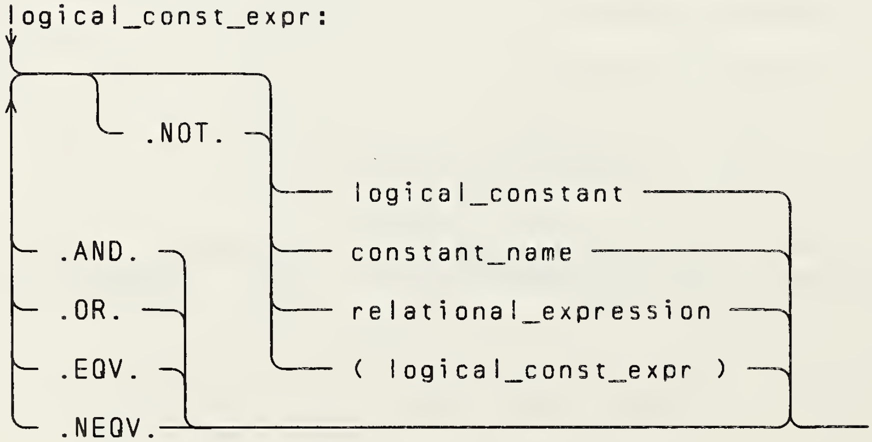
(83) A constant name must be of type character in a character constant expression.

84 logical_expression:



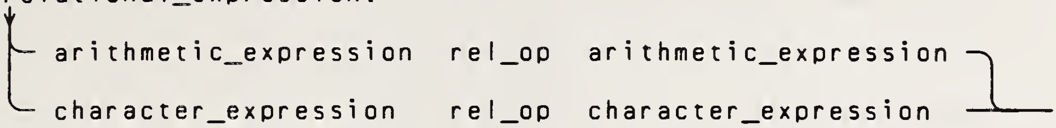
(84) A constant name, variable name, array element name, or function reference must be of type logical in a logical expression.

85 logical_const_expr:



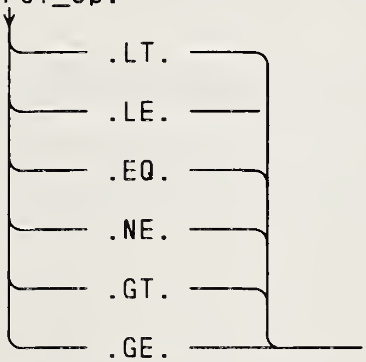
(85) A constant name must be of type logical in a logical constant expression. Also, each primary in the relational expression must be a constant expression.

86 relational_expression:

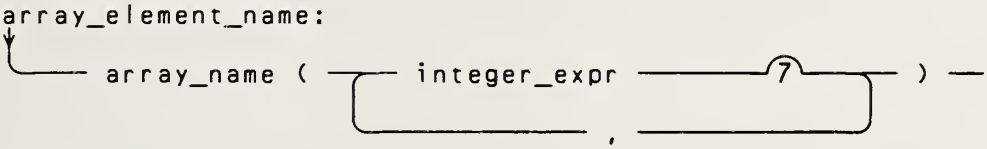


(86) An arithmetic expression of type complex is permitted only when the relational operator is .EQ. or .NE.

87 rel_op:

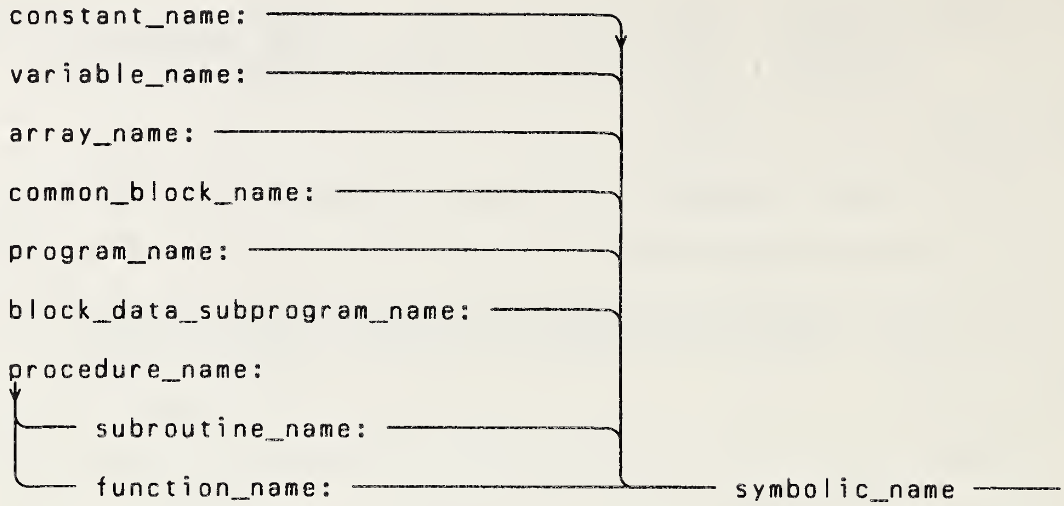


88 array_element_name:

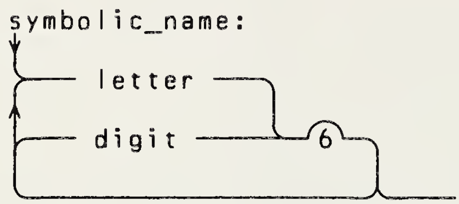


89 substring_name:

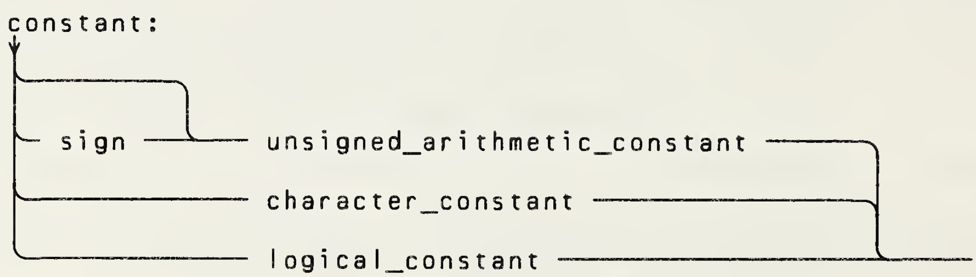
90 91 92 93 94 95 96 97 98



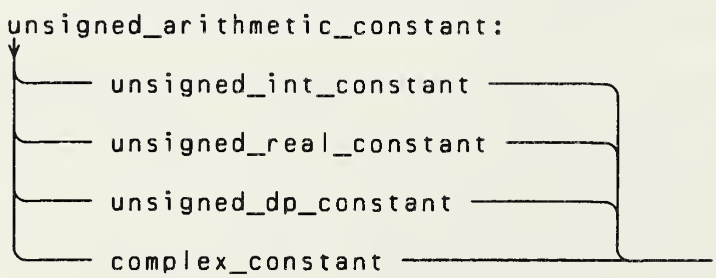
99



100



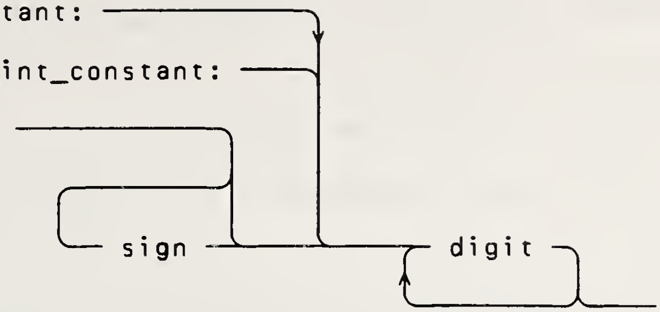
101



102 unsigned_int_constant:

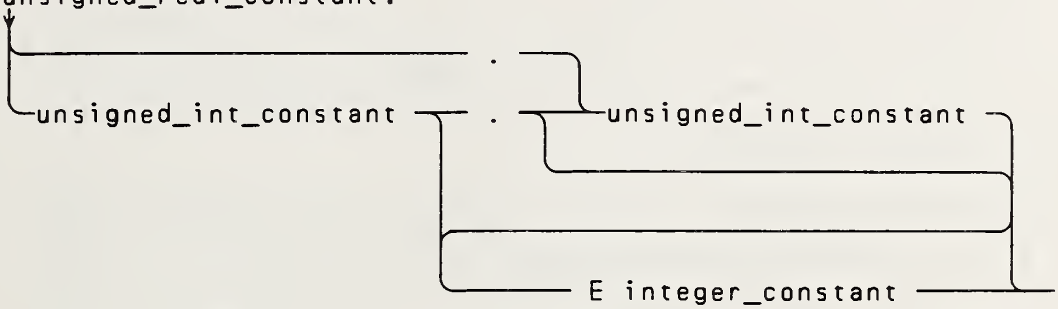
103 nonzero_unsigned_int_constant:

104 integer_constant:

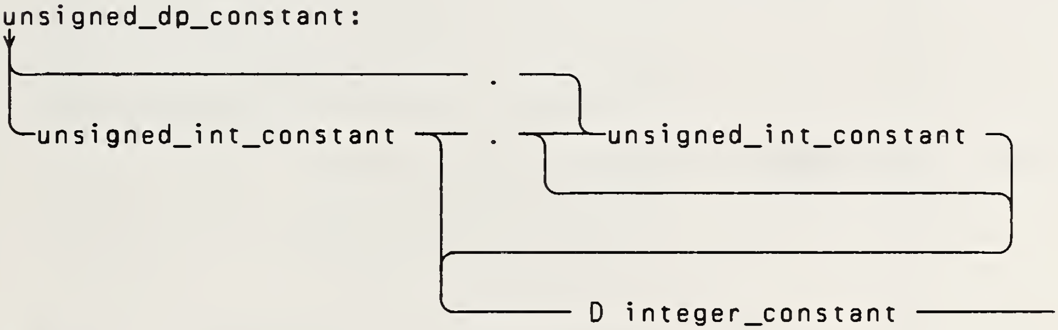


(103) A nonzero, unsigned, integer constant must contain a nonzero digit.

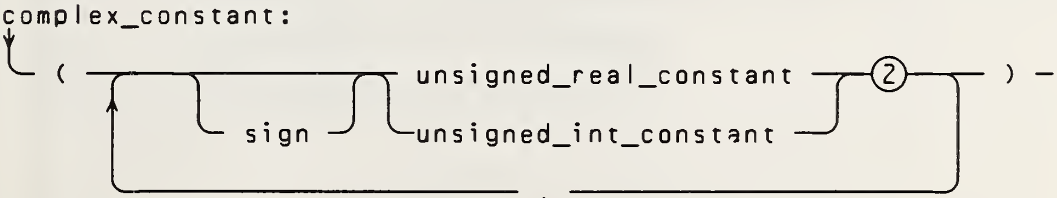
105 unsigned_real_constant:



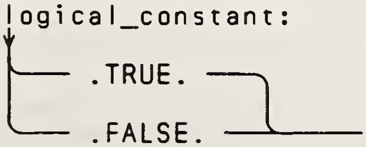
106 unsigned_dp_constant:

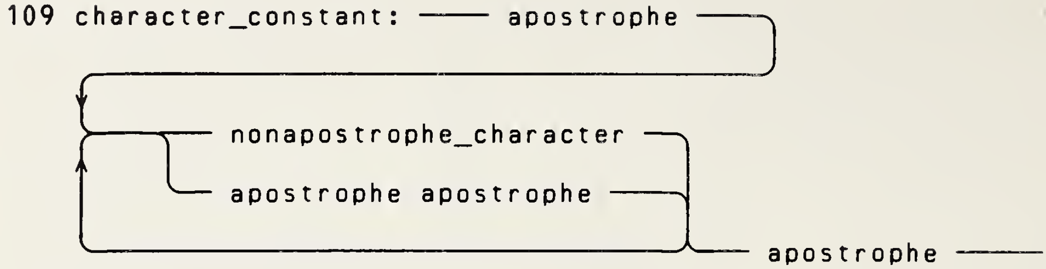


107 complex_constant:

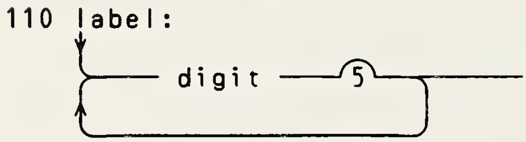


108 logical_constant:

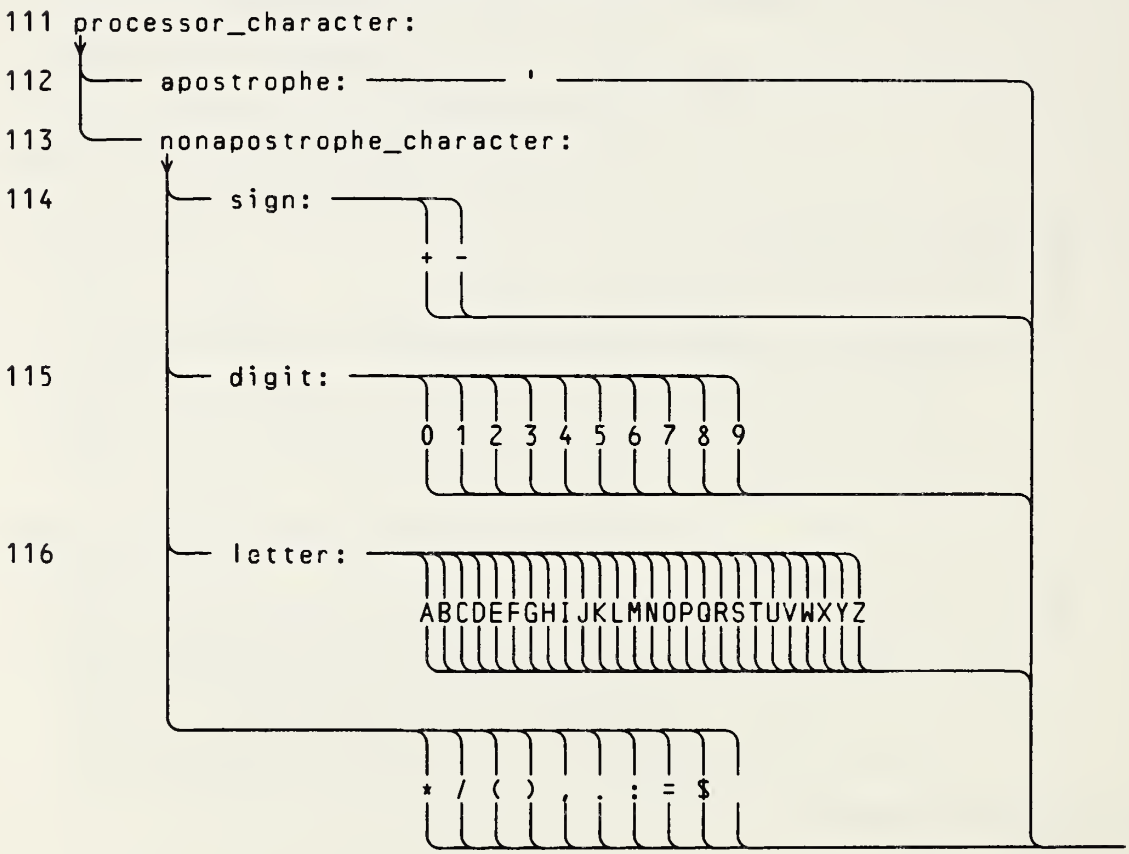




(109) An apostrophe within a data string is represented by two consecutive apostrophes with no intervening blanks.



(110) A label must contain a nonzero digit.



(111) A blank is a processor character. The set of processor characters may include additional characters recognized by the processor.

F3 Cross-Reference Index to Syntax Charts

Def. Item: Reference

Def.	Item: Reference
24	external_statement: 6
60	fmt_specification: 59, 60
57	format_identifier: 46, 47
59	format_specification: 58
58	format_statement: 5
11	function_entry: 9
98	function_name: 11, 20, 25, 70, 73
73	function_reference: 78, 82, 84
10	function_statement: 3
3	function_subprogram: 1
30	goto_statement: 7
69	h: 60
21	implicit_statement: 5
52	inquire_statement: 7
80	int_constant_expr: 22, 28, 80
78	int_real_dp_expr: 34, 40, 49
104	integer constant: 68, 105, 106
77	integer_expr: 32, 47, 50, 56, 72, 88, 89
25	intrinsic_statement: 6
49	io_implied_do_list: 48
48	io_list: 46, 49
68	k: 60
110	label: 5, 29, 31, 32, 33, 34, 40, 47, 50, 51, 52, 55, 57, 71
22	len_specification: 10, 20, 21
116	letter: 21, 99
85	logical_const_expr: 75, 85
108	logical constant: 84, 85, 100
84	logical_expression: 35, 36, 37, 74, 84
35	logical_if_statement: 7
67	m: 60
2	main_program: 1
64	n: 60
113	nonapostrophe character: 109
103	nonzero unsigned int constant: 22, 27, 65
50	open_statement: 7
6	other_specification_statement: 5
23	parameter_statement: 5
43	pause_statement: 7
46	print_statement: 7
96	procedure_name: 11, 13, 24, 71, 73
111	processor character: 69
94	program_name: 8
8	program_statement: 2

INDEX

All references are to page numbers within a section or appendix. For example, 12-4 refers to page 4 in Section 12, and A-1 refers to page 1 in Appendix A.

American National Standards for Information Processing

X3.1-1976 Synchronous Signaling Rates for Data Transmission

X3.2-1970 (R1976) Print Specifications for Magnetic Ink Character Recognition

X3.3-1970 (R1976) Bank Check Specifications for Magnetic Ink Character Recognition

X3.4-1977 Code for Information Interchange

X3.5-1970 Flowchart Symbols and Their Usage in Information Processing

X3.6-1965 (R1973) Perforated Tape Code for Information Interchange

X3.9-1978 FORTRAN

X3.11-1969 Specification for General Purpose Paper Cards for Information Processing

X3.14-1973 Recorded Magnetic Tape for Information Interchange (200 CPI, NRZI)

X3.15-1976 Bit Sequencing of the American National Standard Code for Information Interchange in Serial-by-Bit Data Transmission

X3.16-1976 Character Structure and Character Parity Sense for Serial-by-Bit Data Communication in the American National Standard Code for Information Interchange

X3.17-1977 Character Set and Print Quality for Optical Character Recognition (OCR-A)

X3.18-1974 One-Inch Perforated Paper Tape for Information Interchange

X3.19-1974 Eleven-Sixteenths-Inch Perforated Paper Tape for Information Interchange

X3.20-1967 (R1974) Take-Up Reels for One-Inch Perforated Tape for Information Interchange

X3.21-1967 Rectangular Holes in Twelve-Row Punched Cards

X3.22-1973 Recorded Magnetic Tape for Information Interchange (800 CPI, NRZI)

X3.23-1974 Programming Language COBOL

X3.24-1968 Signal Quality at Interface between Data Processing Terminal Equipment and Synchronous Data Communication Equipment for Serial Data Transmission

X3.25-1976 Character Structure and Character Parity Sense for Parallel-by-Bit Data Communication in the American National Standard Code for Information Interchange

X3.26-1970 Hollerith Punched Card Code

X3.27-1978 Magnetic Tape Labels and File Structure for Information Interchange

X3.28-1976 Procedures for the Use of the Communication Control Characters of American National Standard Code for Information Interchange in Specified Data Communication Links

X3.29-1971 Specifications for Properties of Unpunched Oiled Paper Perforator Tape

X3.30-1971 Representation for Calendar Date and Ordinal Date for Information Interchange

X3.31-1973 Structure for the Identification of the Counties of the United States for Information Interchange

X3.32-1973 Graphic Representation of the Control Characters of American National Standard Code for Information Interchange

X3.34-1972 Interchange Rolls of Perforated Tape for Information Interchange

X3.36-1975 Synchronous High-Speed Data Signaling Rates between Data Terminal Equipment and Data Communication Equipment

X3.37-1977 Programming Language APT

X3.38-1972 (R1977) Identification of States of the United States (Including the District of Columbia) for Information Interchange

X3.39-1973 Recorded Magnetic Tape for Information Interchange (1600 CPI, PE)

X3.40-1976 Unrecorded Magnetic Tape for Information Interchange (9-Track 200 and 800 CPI, NRZI, and 1600 CPI, PE)

X3.41-1974 Code Extension Techniques for Use with the 7-Bit Coded Character Set of American National Standard Code for Information Interchange

X3.42-1975 Representation of Numeric Values in Character Strings for Information Interchange

X3.43-1977 Representations of Local Time of the Day for Information Interchange

X3.44-1974 Determination of the Performance of Data Communication Systems

X3.45-1974 Character Set for Handprinting

X3.46-1974 Unrecorded Magnetic Six-Disk Pack (General, Physical, and Magnetic Characteristics)

X3.47-1977 Structure for the Identification of Named Populated Places and Related Entities of the States of the United States for Information Interchange

X3.48-1977 Magnetic Tape Cassettes for Information Interchange (3.810-mm [0.150-in] Tape at 32 bpmm [800 bpi], PE)

X3.49-1975 Character Set for Optical Character Recognition (OCR-B)

X3.50-1976 Representations for U.S. Customary, SI, and Other Units to Be Used in Systems with Limited Character Sets

X3.51-1975 Representations of Universal Time, Local Time Differentials, and United States Time Zone References for Information Interchange

X3.52-1976 Unrecorded Single-Disk Cartridge (Front Loading, 2200 BPI), General, Physical, and Magnetic Requirements

X3.53-1976 Programming Language PL/I

X3.54-1976 Recorded Magnetic Tape for Information Interchange (6250 CPI, Group Coded Recording)

X3.55-1977 Unrecorded Magnetic Tape Cartridge for Information Interchange, 0.250 Inch (6.30 mm), 1600 bpi (63 bpmm), Phase Encoded

X3.56-1977 Recorded Magnetic Tape Cartridge for Information Interchange 4 Track, 0.250 Inch (6.30 mm), 1600 bpi (63 bpmm), Phase Encoded

X3.57-1977 Structure for Formatting Message Headings for Information Interchange Using the American National Standard Code for Information Interchange for Data Communication Systems Control

X3.58-1977 Unrecorded Eleven-Disk Pack General, Physical, and Magnetic Requirements

X3.60-1978 Programming Language Minimal BASIC

X3.61-1978 Representation of Geographic Point Locations for Information Interchange

X3.62-1979 Paper Used in Optical Character Recognition (OCR) Systems

X3.64-1979 Additional Controls for Use with American National Standard Code for Information Interchange

X3.66-1979 Advanced Data Communication Control Procedures (ADCCP)

X3/TR1-77 Dictionary for Information Processing (Technical Report)

American National Standards Institute, Inc
1430 Broadway
New York, N.Y. 10018